Content-Aware Multimedia Communications

Dissertation
Zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von

Dipl.-Inf. Alexander Eichhorn

Einreichung am 26. Juni 2007
Disputation am 5. Dezember 2007

Gutachter

Prof. Dr.-Ing. habil Winfried Kühnhauser, TU Ilmenau
Prof. Klara Nahrstedt, University of Illinois at Urbana-Champaign
Prof. Dr. Sc. Thomas Plagemann, University of Oslo

urn:nbn:de:gbv:ilm1-2007000398

Content-Aware Multimedia Communications

Dipl.-Inf. Alexander Eichhorn

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor-Ingenieur (Dr.-Ing.)

Faculty of Computer Science and Automation
Technische Universität Ilmenau

Submitted at 26. June 2007
Defended at 5. December 2007

Committee in Charge

Prof. Dr.-Ing. habil Winfried Kühnhauser, TU Ilmenau
Prof. Klara Nahrstedt, University of Illinois at Urbana-Champaign
Prof. Dr. Sc. Thomas Plagemann, University of Oslo

Acknowledgements

This work has its roots in the teaching, help, patience, and inspiration of a great number of people, to whom I wish to express my gratitude.

First of all my deep thanks go to my family who supported me all the times with aid and encouraging words. I highly appreciate their enduring believe in me to successfully finish this thesis. I would also like to thank Dr. Ulrich Klett for convincing me of starting to study computer science at all. Your words and thinking of me encouraged me a lot and I will always keep your advises.

I am deeply indebted to my supervisor Prof. Winfried Kühnhauser for offering me an opportunity to work with the Distributed Systems Group at TU Ilmenau. I would like to thank you for the lasting support throughout my studies, all the guiding, discussions and cooperation. Even if the duration has been longer than originally planned, I appreciate the belief you have expressed in my ability to finish the work and for providing me with the required funding.

A lot of people have contributed to this work in various ways. Many of the ideas presented herein evolved during discussions with my friends, colleagues and students. I would particularly like to thank my friend Andreas Franck for the numerous enlightening discussions during the last years and for his detailed, insightful comments on nearly every page of this work. I would also like to thank my friends and colleagues Dr. Thorsten Strufe, Dr. Carsten Behn and Mario Holbe for their much appreciated ideas, support, and feedback as well as Katja Wolf for their great administrative support. In addition, I would like to thank all colleagues in the Institute of Computer Science at TU Ilmenau for the pleasant working environment.

Writing this thesis would not have been possible without the efforts of many friends and students who contributed to the development of the Noja platform. Special thanks go to Carsten König, Christian Brien, Sebastian Kühn, Alexander Hans, Christoph Böhme, Alexander Senier, Michael Roßberg, Melanie Friedrich, Holger Möller, Florian Kriener, Jens Dönhoff, Tino Jungebloud, Ives Steglich, and the numerous other people who worked for the Noja project.

Finally, I like to express my deep gratitude to all my friends, especially my flat-mates Sebastian Daume, Klemens Göbel, Stefan Hertwig, Janosch Lax, Jana Kludas, Alexander Schultz, Carsten Riesel, Sandy Stehr and Oliver Hitzegrad, all friends who lived in the K4, and all the others I forgot to mention. Although I was often busy and non-present, neither physically nor mentally, I highly appreciate the mental support I received from all of you. I am truly glad that all of you have influenced my way of living and thinking.

Abstract

The demands for fast, economic and reliable dissemination of multimedia information are steadily growing within our society. While people and economy increasingly rely on communication technologies, engineers still struggle with their growing complexity.

Complexity in multimedia delivery originates from several sources. The most prominent is the unreliability of packet networks like the Internet. Recent advances in scheduling and error control mechanisms for streaming protocols have shown that the quality and robustness of multimedia delivery can be improved significantly when protocols are aware of the content they deliver. However, the proposed mechanisms require close cooperation between transport systems and application layers which increases the overall system complexity. Current approaches also require expensive metrics and focus on special encoding formats only. A general and efficient model is missing so far.

This thesis presents efficient and format-independent solutions to support cross-layer coordination in system architectures. In particular, the first contribution of this work is a generic dependency model that enables transport layers to access content-specific properties of media streams, such as dependencies between data units and their importance. The second contribution is the design of a programming model for streaming communication and its implementation as a middleware architecture. The programming model hides the complexity of protocol stacks behind simple programming abstractions, but exposes cross-layer control and monitoring options to application programmers. For example, our interfaces allow programmers to choose appropriate failure semantics at design time while they can refine error protection and visibility of low-level errors at run-time.

Based on examples we show how our middleware simplifies the integration of stream-based communication into application architectures. An important result of this work is that despite cross-layer cooperation, neither application nor transport protocol designers experience an increase in complexity. Application programmers can even reuse existing streaming protocols which effectively increases system robustness.

Kurzfassung

Der Bedarf unsere Gesellschaft nach kostengünstiger und zuverlässiger Kommunikation wächst stetig. Während wir uns selbst immer mehr von modernen Kommunikationstechnologien abhängig machen, müssen die Ingenieure dieser Technologien sowohl den Bedarf nach schneller Einführung neuer Produkte befriedigen als auch die wachsende Komplexität der Systeme beherrschen. Gerade die Übertragung multimedialer Inhalte wie Video und Audiodaten ist nicht trivial. Einer der prominentesten Gründe dafür ist die Unzuverlässigkeit heutiger Netzwerke, wie z.B. dem Internet. Paketverluste und schwankende Laufzeiten können die Darstellungsqualität massiv beeinträchtigen. Wie jüngste Entwicklungen im Bereich der Streaming-Protokolle zeigen, sind jedoch Qualität und Robustheit der Übertragung effizient kontrollierbar, wenn Streamingprotokolle Informationen über den Inhalt der transportierten Daten ausnutzen.

Existierende Ansätze, die den Inhalt von Multimediadatenströmen beschreiben, sind allerdings meist auf einzelne Kompressionsverfahren spezialisiert und verwenden berechnungsintensive Metriken. Das reduziert ihren praktischen Nutzen deutlich. AuSSerdem erfordert der Informationsaustausch eine enge Kooperation zwischen Applikationen und Transportschichten. Da allerdings die Schnittstellen aktueller Systemarchitekturen nicht darauf vorbereitet sind, müssen entweder die Schnittstellen erweitert oder alternative Architekturkonzepte geschaffen werden. Die Gefahr beider Varianten ist jedoch, dass sich die Komplexität eines Systems dadurch weiter erhöhen kann.

Das zentrale Ziel dieser Dissertation ist es deshalb, schichtenübergreifende Koordination bei gleichzeitiger Reduzierung der Komplexität zu erreichen. Hier leistet die Arbeit zwei Beträge zum aktuellen Stand der Forschung. Erstens definiert sie ein universelles Modell zur Beschreibung von Inhaltsattributen, wie Wichtigkeiten und Abhängigkeitsbeziehungen innerhalb eines Datenstroms. Transportschichten können dieses Wissen zur effizienten Fehlerkontrolle verwenden. Zweitens beschreibt die Arbeit das Noja Programmiermodell für multimediale Middleware. Noja definiert Abstraktionen zur Übertragung und Kontrolle multimedialer

Ströme, die die Koordination von Streamingprotokollen mit Applikationen ermöglichen. Zum Beispiel können Programmierer geeignete Fehlersemantiken und Kommunikationstopologien auswählen und den konkreten Fehlerschutz dann zur Laufzeit verfeinern und kontrollieren.

Contents

1 Introduction **1**

1.1 Challenges in Multimedia Communications 2

1.2 Content-Aware Media Streaming 5

1.2.1 Design Philosophy 5

1.2.2 Assumptions and Objectives 7

1.3 Contributions of this Thesis 8

1.4 Dissertation Outline 10

2 Background on Multimedia Streaming Systems **13**

2.1 Distributed Multimedia Applications 14

2.1.1 Application Classes 14

2.1.2 Application Architectures and Topologies 19

2.2 Streaming Protocol Standards 24

2.2.1 MPEG Transport Specifications 24

2.2.2 Multimedia Protocols for the Internet 26

2.2.3 Streaming in Mobile and Wireless Networks 28

2.3 Network-adaptive Multimedia Streaming 29

2.3.1 Sources and Effects of Network Errors 29

2.3.2 Error-Resilient Signal Encoding 31

2.3.3 Adaptive Error Control 35

2.3.4 Adaptive Rate and Congestion Control 38

2.3.5 Buffer Management and Synchronisation 40

2.4 Conclusion 41

3 Related Work **43**

3.1 Stream-based Programming Abstractions 43

3.2 Open Middleware Platforms and QoS-aware Middleware . 45

3.3 Multimedia Middleware Platforms 47

3.4 Closely related Multimedia Middleware 49

3.5 Content-Awareness in Streaming Protocols 50

4 A Framework for Content-Aware Media Streaming 53
4.1 Content-Aware System Layers 54
4.2 Objectives and Challenges 56
4.3 Structural Properties of Media Streams 58
4.3.1 Quality and Distortion 58
4.3.2 Dependency Relations 60
4.3.3 Visibility and Predictability of Structure 65
4.4 Modelling Dependency Relations 67
4.4.1 Type-based and Object-based Dependency 67
4.4.2 Dependency-Graph Operations 69
4.5 Type-Graph Attributes 71
4.5.1 Dependency Rules 72
4.5.2 Type Attributes 75
4.5.3 Group Semantics 77
4.6 Object-Graph Attributes and Operations 77
4.6.1 Object Graph Decoration 80
4.6.2 Structure and Importance Prediction 82
4.6.3 Dependency Validation 87
4.6.4 Importance Estimation 89
4.7 Dependency Model Implementation 93
4.7.1 Dependency Description Language 94
4.7.2 H.264/AVC Video Stream Example 99
4.7.3 Dependency Validation Service 103
4.7.4 Embedding the DVS into System Layers 108
4.8 Limitations of the Dependency Model 109
4.8.1 Estimation Accuracy 109
4.8.2 Horizon Size and Visibility 111
4.8.3 Loss-Resilience 115
4.9 Experimental Evaluation 116
4.9.1 Properties of Selected Video Sequences 117
4.9.2 Accuracy of Dependency-based Importance 124
4.9.3 Effects of Limited Horizon Size 138
4.9.4 Effects of Packet Loss 143
4.9.5 Performance Benchmarks 146
4.10 Conclusion 150

5 Noja: A Content-Aware Streaming Middleware Platform 153
5.1 Design Principles 154
5.1.1 Assumptions on Target Environments 154
5.1.2 Interaction Model 155

5.1.3 Transparency and Cross-Layer Issues 157
5.1.4 Deliberate Omissions 160
5.2 Novelty of the Noja Middleware Platform 162
5.3 Middleware Abstractions and Operations 163
5.3.1 Communication Abstractions 163
5.3.2 Binding Semantics 165
5.3.3 Port Configuration 170
5.3.4 Binding Ports 173
5.3.5 Data Transfer and Stream Unit Labelling 176
5.3.6 Application-Level Signalling 178
5.3.7 Flow-Control 178
5.3.8 Stream Synchronisation 180
5.3.9 Monitoring and Performance Feedback 182
5.4 Middleware Implementation 182
5.4.1 Middleware Architecture Overview 183
5.4.2 Application Programming Interface 186
5.4.3 Binding Establishment 195
5.5 Application Examples 202
5.5.1 Case Studies for Coordinated Error Protection . . 202
5.5.2 A Newscast Application Scenario 204
5.6 Conclusion 208

6 Conclusion and Future Work 211
6.1 Achievements 212
6.2 Open Questions 213
6.3 Future Research Directions 214

A Dependency Description Language 217

Bibliography 221

List of Tables

2.1 Requirements of different streaming application classes. . 15
2.2 Requirements (continued). 17

4.1 NAL unit types in H.264/AVC and H.264/SVC. 100
4.2 Dependency-Related attributes in data unit labels. 105
4.3 The set of standard video sequences used in our experiments. 118
4.4 The set of BBC video sequences used in our experiments. 119
4.5 NAL Units per video frame. 122
4.6 NAL Units per video frame (continued). 123
4.7 NAL Units per epoch. 123
4.8 Performance of importance estimation for standard sequences. 128
4.9 Performance of importance estimation for BBC sequences. 129
4.10 Hypothetical packet scheduler performance analysis. . . . 149

5.1 Port operations 171
5.2 Port properties. 173
5.3 Binding properties. 175
5.4 Label attributes 177
5.5 Synchronisation properties. 181
5.6 Monitoring information. 183
5.7 QoS configuration example 190
5.8 Selected message types of the Noja signalling protocol. . . 198

List of Figures

1.1 Building blocks of the middleware vision. 6
1.2 Dissertation overview. 10

2.1 Video coding system model. 22
2.2 Data-paths and delays in streaming transport protocols. . 23
2.3 Encoder rate control model. 39

4.1 Examples of dependency structures. 61
4.2 Dependecy structures in fragmented streams. 61
4.3 Static and dynamic importance distribution example. . . 63
4.4 Visibility of data units and dependency relations. 66
4.5 Type-graph for MPEG-like streams. 75
4.6 Implicit decoration, based on type information only. . . . 81
4.7 Limits of implicit decoration and benefits of explicit decoration. 83
4.8 Type prediction example. 86
4.9 Importance estimation vs. dependency validation processes. 88
4.10 Cross-Layer design using the content-awareness framework. 108
4.11 MSE distortion vs. importance. 110
4.12 Statistical plots of distortion and importance in Foreman. 110
4.13 Importance distribution maps. 112
4.14 Effects of loss on dependency chains. 116
4.15 NAL unit size distribution. 121
4.16 NAL unit size and number in error-resilient sequences. . . 121
4.17 Epoch size distribution in error-resilient sequences. 124
4.18 Correlation between importance and distortion for standard sequences. 130
4.19 Correlation between importance and distortion for BBC sequences. 130
4.20 Statistical plots for selected sequences. 131
4.21 Correlation between importance and distortion for error resilient sequences. 133

4.22 Correlation between importance and distortion at different bitrates. 134
4.23 Statistical plots for error-resilient sequences. 135
4.24 Correlation between importance and distortion at different GOP-sizes. 136
4.25 Statistical plots for sequences with and without B-frames. 137
4.26 Impact of horizon size on importance inversions. 140
4.27 Impact of GOP size on importance inversions. 142
4.28 Impact of loss rate on importance inversions. 144
4.29 Impact of GOP size on importance inversions under loss. . 145
4.30 Decoration and estimation performance for multiple dependency patterns. 147
4.31 Decoration and estimation performance at different GOP-sizes. 148

5.1 The Noja communication model. 156
5.2 State charts of OUT-ports and IN-ports. 179
5.3 Overview of the Noja middleware architecture. 184
5.4 Distributed synchronisation group conflicts 192
5.5 Zero-copy buffers to store stream units. 193
5.6 Binding establishment procedure. 195
5.7 Live newscast example. 204
5.8 Newscast example system model. 206

A.1 DDL syntax diagrams. 218

Listings

4.1 Dependency description for a MPEG-like video stream. . 95
4.2 H.264/AVC dependency description. 102
4.3 Dependency Validation Service API. 104
5.1 Object Adaptor API (selected operations). 188
5.2 QoS performance specification and the `bind()` operation. 189
5.3 Synchronisation interfaces. 191
5.4 Typical use-case of the port API. 207
A.1 EBNF of the Dependency Description Language. 217
A.2 Full H.264 dependency description. 219

Chapter One

Introduction

The general precept of any product
is that simple things should be easy,
and hard things should be possible.

(Alan Kay)

The demands for fast, economic and reliable dissemination of multimedia information are steadily growing within our society. With the availability of digital networks, efficient signal compression algorithms and powerful end-user devices, human communication shifted from traditional analogue media like letters, newspapers, wired telephones, radio and television broadcasts to digital technologies and new types of communication.

Mobile phones quickly changed the behaviour of personal communications by decoupling availability from physical location. Novel applications such as audiovisual conferencing and collaboration systems appeared, and today we are surrounded by new types of applications, like Instant Messaging, Blogs, Podcasts and Web-based portals like YouTube, flickr and mySpace that enable large networked communities to share information, opinions and experiences.

This trend is likely to continue as the availability of appropriate network infrastructures as well as cheap, flexible and powerful communication devices increases. Equipped with new kinds of sensory inputs, such as location and environmental sensors, new devices will enable new types of applications and user interfaces, fostering remote interactions between people, connecting different cultures and breaking down language barriers. One flipside of this evolution is that our society puts more and more reliance on information technology, while engineers still struggle with their growing complexity.

In reaction, this thesis contributes a middleware platform that focuses on the end-to-end transport of real-time multimedia streams. The middleware hides the complexity of streaming protocols behind simple programming abstractions and exposes sophisticated control and monitoring options to application programmers. In order to raise the efficiency and robustness of stream delivery against communication failures, this work is based on a theoretical foundation for content-awareness that allows transport layers to reason about the properties of the data they deliver.

This introduction discusses main challenges in the field on multimedia communications, identifies requirements on an appropriate middleware platform, summarises the central contributions of this thesis and presents an outline of it's organisation.

1.1 Challenges in Multimedia Communications

Building distributed multimedia systems requires close coordination between several disciplines. Besides signal processing, human interface design, content-analysis, presentation and many others a prominent discipline is communication. Communication deals with protocols, network infrastructures, and also programming environments. Because most of the emerging application scenarios will continue to be distributed and resource demanding, this raises demands for robust, efficient and scalable communication systems. While research on compression algorithms and computer networks has led to significant improvements in terms of quality, scalability and robustness of multimedia delivery, the proposed mechanisms also increased the overall system complexity.

The main objective of multimedia communication is to achieve the optimal signal reconstruction quality for end-users despite of network failures. In general, compressed data units in media streams have delivery deadlines, may have variable size, and they are unequally important to signal reconstruction. While delivery over Quality-of-Service (QoS) networks benefits from predictable bandwidth, delays and error bounds, best effort networks show an unpredictable performance. Here, adaptive rate- and error control algorithms are required to optimally protect a data stream by allocating the limited bandwidth to media payload and error control data under given quality and deadline constraints. Frameworks for determining the relative importance of data units in media streams already exist [1–3], but they suffer from high computational complexity which is problematic for resource-constrained systems.

When rate and error performance of the delivery channel are known at encoding time, joint approaches that consider content and network conditions can directly adapt the encoding scheme. When, in contrast, the channel characteristics are unknown during encoding, it is desirable to employ scalable encoding schemes and let the sender application later decide which parts of the stream to actually transmit, which parts to repair after loss and which to drop. Although scalable encoding introduces additional storage and processing overheads, it is regarded as the key technology to enable large-scale real-time Internet streaming.

Regardless of encoding formats and adaptation policy, adaptation schemes require some form of coordination between application and transport layers to exchange information about channel characteristics and stream properties. Current protocol architectures complicate the necessary information flows since their interfaces are designed for information hiding. Cross-Layer protocol architectures [4] are intended to remedy these deficiencies, but due to the lack of clear interface designs, current approaches decrease interoperability rather than providing generic solutions.

Current protocol standards for streaming transport and signalling, such as RTP [5], SIP [6] and SDP [7], provide basic services for stream delivery and session management, including for example payload identification, sequence numbering, time stamping and feedback. These frameworks are already complex due to their numerous features and operation modes, and they intentionally lack support for control algorithms. Congestion control and quality-of-service negotiation, proper fragmentation, error control and scaling is left to other protocols (such as the DCCP [8] or RSVP [9]), but mostly delegated to the application level. While many solutions for fragmentation, scaling and sophisticated error control algorithms do exist, they are fixed to special encoding formats or network environments. A flexible framework that is reusable in different application scenarios is missing so far.

Given the spectrum of encoding formats and protocol mechanisms, delivering partially loss-resilient media streams in a robust and efficient way over best-effort packet networks requires an amount of expert knowledge, that could not be expected from a typical application programmer. Middleware platforms are known as a comfortable and appropriate tool for constructing large-scale distributed applications. While supporting typical communication patterns and application-centric communication semantics they hide heterogeneity and complexity of transport protocols and system-level interfaces. A ordinary programmer is relieved from complex and error prone tasks while an advanced programmer can selectively configure and extend essential behaviour. Thus a well-designed middleware can help to construct dependable and efficient streaming systems. Although there are many challenges, this thesis contributes solutions to three requirements we regard most important for the design of an integrating stream-based communication middleware:

Complexity in multimedia communication originates from many sources, such as the myriad of signalling protocol features, the heterogeneity of end-user devices and network infrastructures in performance, reliability and support of Quality-of-Service and back channels. Moreover, the diverse error resilience features of encoding formats as well as the multitude of error control protocols, buffer management algorithms and synchronisation schemes introduce additional complexity for application programmers who rather prefer to integrate simple control and transport mechanisms into their applications. In order to decrease complexity and relieve programmers from such error-prone and time-consuming tasks, simple programming abstractions and reusable protocol frameworks are required. This thesis defines such abstractions for the domain of real-time media streaming.

Adaptation is necessary to deliver scalable and partially loss-resilient media streams over unreliable best-effort networks like the Internet. Adaptation requires cooperation between layers to exchange information that is hidden by traditional protocol layering. Techniques for unequal error protection, for example, require knowledge about the properties of application-level data units to infer their importance, while scalable and rate-distortion optimised streaming as well as joint source-channel coding depend on the current channel performance. Regardless of the location of adaptation strategies, access to this information must be available, but a cross-layer design must avoid negative impacts on the overall complexity and stability of a system. This thesis identifies relevant stream properties and defines interfaces to exchange them in order to enable cooperative error control and scheduling between network-adaptive applications and content-aware system layers.

Flexibility is a key challenge for a multimedia middleware platform. It is unlikely that a single multimedia format or a single transport protocol will fit all individual requirements of streaming applications across diverse networking technologies and user devices. Hence, a middleware architecture must support different network protocols, different encoding formats, diverse application topologies, interaction patterns and quality requirements. The middleware platform proposed in this thesis contains an open and extensible architecture to meet these requirements.

1.2 Content-Aware Media Streaming

The abstraction of communication is the primary objective of middleware platforms in distributed computing. A great variety of multimedia middleware concepts has been investigated in recent research projects, such as streaming extensions for object-based middleware [10, 11], QoS-aware resource management [12–16], new programming abstractions [17], synchronisation and buffer control [18], and application-level management and adaptation [19, 20].

While some solutions address isolated problems only, all of them contain useful algorithms, protocol frameworks and programming abstractions where we can build upon. In particular, we extend the existing work by special interfaces to enable cross-layer cooperation between transport protocols and applications. Because current solutions are limited to special encoding formats, selected application scenarios or network environments this thesis proposes a novel middleware design that avoids such restrictions.

1.2.1 Design Philosophy

The work in this thesis follows the central vision of efficient and easy-to-use interfaces that seamlessly integrate with existing and novel application scenarios, networking technologies and encoding formats. Our middleware platform is based on a generic programming model that selectively hides low-level details of protocols, but exposes relevant control and feedback options to network-adaptive applications. To benefit from content-specific protocol processing the interfaces should also allow to pass information from an application across layer boundaries down into the protocol stack.

Feedback would help network-adaptive applications to respond to variations in available bandwidth or channel error performance by adjusting their transfer rate, encoding schemes and error protection strategies.

Content-aware transport layers on the other hand can consider properties of the data they deliver in strategic decisions. A protocol can, for example, protect important data stronger and avoid forwarding data after a specified delivery deadline.

Adjusting application-layer processing and protocol processing requires coordination to gain efficiency and increase robustness. It is therefore necessary to (1) select relevant properties of data units that are worth exploiting at lower system layers, (2) select relevant characteristics of lower system layers that are relevant for adaptive applications, and (3) find interaction patterns and interfaces that allow for efficient metadata passing and coordination across system layers without breaking layer assumptions. In order to build the envisioned middleware platform, we regard the following four steps as essential (see figure 1.2.1):

First, a **Content-Awareness Model** is required to universally express and reason about properties of data streams such as error resilience, dependency, deadlines, data rate and the quantity of information contained in each data unit. The model should allow mathematical analysis, be computationally tractable and it should use information that is already available or that may be generated at low overheads to be efficiently implementable. Based on this model, we will define a set of meta-information that can be universally shared with transport protocols.

Second, **Streaming Protocols** that can utilise content metadata are required. Such protocols can use the additional information to adjust their error control and packet scheduling mechanisms, perform optimal error protection, packet drop and packet forwarding in different appli-

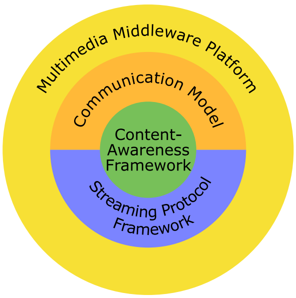

Multimedia Middleware Platform
Communication Model
Content-Awareness Framework
Streaming Protocol Framework

Fig. 1.1 : *Building blocks of the middleware vision.*

cation scenarios. Because different applications have different optimisation constraints and reliability requirements, different configurations are likely. A large spectrum of mechanisms does already exist. Hence, this thesis does not develop novel protocols or error-protection schemes. Instead, it defines metadata passing mechanisms break the tight coupling between existing protocols and specific applications and codecs.

Third, a **Communication Model** that defines simple programming abstractions and flexible communication semantics is required. This model should hide details of transport and signalling protocols from programmers, but expose cross-layer coordination facilities. This model should define operations for session handling, signalling and real-time stream delivery, and it should allow programmers to choose between different communication semantics to tune the behaviour of operations. Semantics should define how operations behave in the presence of communication failures, interaction patterns and application topologies.

Forth, a **Middleware Platform** is required to integrate the above models into a single building block that can be used by application programmers to delegate all communication-related tasks. The middleware should implement the communication model and provide a runtime environment for protocols. It should define interfaces to control and monitor stream delivery and it may contain support for automated selection of protocols. We expect this middleware to considerably reduce the overall system complexity of distributed multimedia streaming because protocols can be reused in different application scenarios without requiring an application programmer to maintain intimate knowledge of protocol details and networks. This will increase efficiency and robustness of applications and foster the innovation of novel applications.

1.2.2 Assumptions and Objectives

In the context of this thesis, multimedia streaming is seen as the unidirectional delivery of a continuous sequence of time-sensitive data units between a single sender and one or more receivers over a packet network. We assume that applications may generate live-encoded streams or deliver offline-encoded streams over unicast, multicast or broadcast topologies. Because back-channels may be unavailable, feedback is not necessarily part of this definition. When available, feedback may be used by protocols and the application to adjust stream delivery. In particular, we do not consider relations between multiple streams that share a common network path or streams that are multiplexed.

Because packet networks are vulnerable to contention and packet loss, media streams may be affected by unpredictable loss, bandwidth variations and variable transmission delays. We assume that an application may either be network-aware or not and it may have reliability and timing requirements that change over time. Host devices may have limited resources and may be concurrently connected to multiple networks for multi-path streaming or network handover. We further assume that the network may or may not support multicasting and different qualities of service with predictable performance or priority levels.

These assumptions suite a variety of real-world scenarios of general interest, such as large-scale broadcast distribution of stored or live encoded streams, on-demand delivery of pre-encoded streams, and delivery of interactive streams over wired and wireless packet networks, over broadcasting and multicasting networks of telephone carriers, and over dedicated networks with predictable QoS or over the best-effort Internet.

In conclusion, the general objectives behind the work presented in this thesis are:

- foster the seamless integration of real-time multimedia streaming into application architectures,
- be independent of stream formats, application topologies, and interaction patterns to reuse protocol implementations,
- support simple programming abstractions and communication semantics to hide system complexity, location and network diversity,
- expose relevant channel properties and dynamics to network-adaptive applications,
- express, share and exploit error-resilience properties of media streams to increase efficiency and robustness of transport protocols.

1.3 Contributions of this Thesis

The main contributions of this thesis are:

1. A generic framework for **content-awareness** that enables system layers to access and track properties of media streams such as dependency, importance and deadlines with a low computational complexity.

2. A **cross-layer design philosophy** that uses hints as the general means to exchange meta-data in combination with data units. Hinting extends layered system architectures without violating layering assumptions. It can be used to efficiently implement the integrated layer processing (ILP) concept.

3. The **Noja programming model** for cross-layer coordinated multimedia streaming that defines simple programming abstractions, extensible binding semantics and flexible communication interfaces.

4. The **Noja middleware platform** which implements the novel programming model. The middleware selectively hides details of streaming and signalling protocols and exposes relevant feedback to network-adaptive applications.

The presented work deliberately omits to define special rate-control, error-control or scheduling schemes. It is also not our intention to design another streaming protocol because many proper solutions do already exist. Instead, we identify a generic set of meta-data attributes about a media stream which is required by existing protocols. Our middleware interfaces will enable applications to efficiently propagate this information to transport layers. we do also not consider coordination with protocols below the transport layer although this may gain further efficiency benefits. In contrast, we focus on problems that require an end-to-end view of the system architecture.

Furthermore this work does not propose a new quality-of-service framework or policies to enforce real-time constraints nor mechanisms for QoS negotiation. Instead, we build upon results of recent QoS research. We do also not propose a novel adaptation framework or policies to adaptively control resource consumption of applications and streaming protocols. We do, however, provide basic adaptation support at the interface level by allowing an application to monitor channel characteristics and execution times. Existing adaptation frameworks can build upon this feature.

We do also not consider security and privacy issues, but we believe that proper mechanisms can be easily integrated into our middleware using familiar concepts. Where appropriate, we make a side-note on how protocols and mechanisms for authentication, authorisation and key exchange could be integrated.

1.4 Dissertation Outline

Figure 1.4 displays a graphical overview of the thesis organisation. Chapter 2 discusses relevant background information on application requirements, multimedia communication protocols and advances in the area of multimedia delivery over best-effort packet networks. We identify general problems and shortcomings of current communication technologies and draw some general conclusions which will guide the following design decision.

In chapter 3 we will briefly review work that is related to this thesis. Because our work combines approaches from different research communities, the multimedia and systems community and the signal processing community, we focus on recent developments in both areas. This comprises programming models for stream-based interactions and middleware platforms for multimedia systems. Furthermore, related work includes models and metrics to express content-specific attributes of media streams as well as cross-layer protocol design issues.

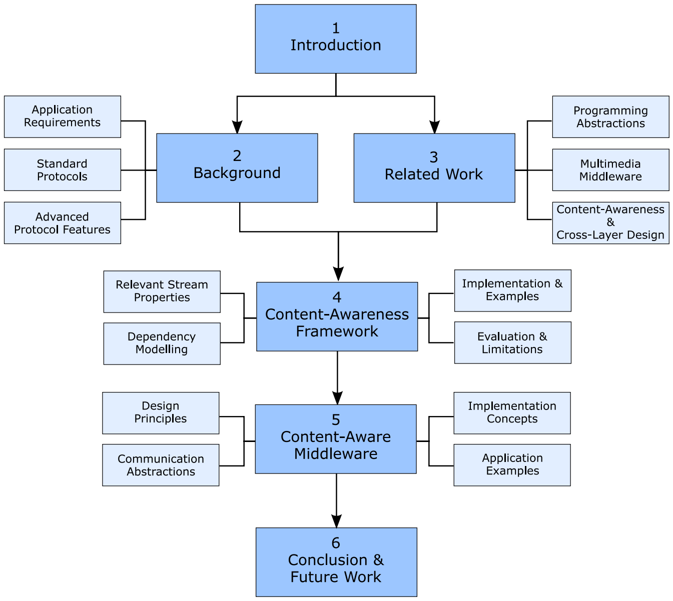


Fig. 1.2 : *Dissertation overview.*

In chapter 4 we develop a generic content-awareness framework which allows us to express and reason about properties of media streams without bothering about encoding formats. We first identify properties that are relevant for transport layer tasks, such as dependencies and importance. We then propose a model for robust and efficient tracking of dependency relations and importance estimation. We present an implementation of our framework and provide examples to show how it integrates with system layers. Finally, we statistically compare the estimation accuracy of our framework to existing solutions and investigate several effects that may influence its precision.

Based on this framework we develop a content-aware middleware platform in chapter 5. Starting from general design principles, we develop common communication abstractions and semantics that are intended to fit a broad range of streaming application. We then present relevant implementation details of our middleware and show, how applications can use it to coordinate network-adaptive streaming with content-aware transport layers.

Chapter 6 finally concludes with the main results and gives an outlook on open research topics in the context of content-aware system layers.

Chapter Two

Background on Multimedia Streaming Systems

Research is to see what everybody
else has seen, and think what
nobody else has thought.

(Szent-Györgyi Nagyrápolt)

Digital multimedia systems in general and multimedia communication systems in particular are an active research area for more than 20 years. Since the late 1980's researchers and engineers have tremendously improved the efficiency of signal compression technologies, the robustness of communication mechanisms, the quality-of-service support in networks and operating systems, as well as the scalability of system architectures. There are several comprehensive surveys on recent developments [3, 21–26].

The general objective of multimedia communications is the delivery of multimedia data at the optimal quality and under acceptable costs. With the pervasive availability of packet networks, the trend in multimedia communications shifts from expensive QoS-controlled networks towards stream delivery over wireless packet networks and the best-effort Internet. Fundamental challenges in these environments remain the uncontrollable end-to-end delay, the unknown and time-varying availability of bandwidth, the different types of transmission errors, and the demands for mechanisms to ensure fairness, security and privacy [23].

In this section we will briefly discuss relevant types of multimedia applications and their requirements on communication systems, transport protocols and advanced coding algorithms. For brevity we neglect fairness, security and privacy issues.

2.1 Distributed Multimedia Applications

Although the diversity of multimedia applications is large, all applications that handle continuous media streams share some common characteristics in topologies, interaction patterns and quality requirements. This section summarises common types of streaming applications and their communication requirements. We will later use this knowledge to design proper abstractions and communication semantics for our middleware platform.

2.1.1 Application Classes

Multimedia systems integrate several types of non-continuous and continuous data such as text, images, textures, speech, music, video, haptics, location and other sensory inputs. Depending on device capabilities and application purpose the data is processed and transmitted in different qualities and under different reliability and real-time constraints.

Although the solutions proposed in this thesis are generally applicable to all applications that exchange continuous data streams, we are in particular interested in distributed streaming applications which make use of continuous audio-visual data types such as video, audio and voice data (see table 2.1 and 2.2). We identified the following main application classes which use audio-visual content:

Broadcasting Applications Broadcasting is used to efficiently deliver popular content simultaneously to many receivers. The content is either pre-encoded when a program is produced offline or live encoded in real-time when live events such as news and sports events are broadcasted. Users demand high resolutions and high quality from broadcasting services.

Broadcasting applications typically use one-way channels with high and constant bandwidth. Feedback is often undesired and channel conditions are unknown. Although this ensures scalability, error protection schemes must prepare streams for the worst-case receiver conditions to provide adequate quality to all receivers. Complexity of encoding algorithms is allowed to be high because resources are available at the sender side, while decoder complexity must be low, in particular for mobile devices. The total end-to-end delay is fixed and generally less important than short startup and

Application Class	**Broadcasting**	**On-Demand**	**Editing**
E2E Delay	moderate	long	low
Startup Delay	low	moderate	low
Rebuffering	no	yes	yes
Resolution	high	high	very high
Coding	efficient, loss-resilient, low-complexity decoder	efficient, loss-resilient, low-complexity decoder	loss-less, random access
Error Control	FEC	FEC + ARQ	ARQ
Channel	controlled exclusive	uncontrolled shared	uncontrolled exclusive
Topology	broadcast no back-channel	uni/multicast back-channel	unicast back-channel

Tab. 2.1 : *Requirements of different streaming application classes [27, 28].*

channel switching delays. Fixed channel bandwidth and switching requirements limit predictive encoding schemes to constant bitrates and reasonably frequent self-containing data units (I-frames) to generate closely spaced switching points. Although stream data experiences a fixed delay and arrives in-order, a stream may still be corrupted by bit and burst errors.

Examples of already deployed broadcasting systems are Digital Video Broadcasting[1] (DVB) for television, Digital Audio Broadcasting (DAB) [29] for radio, and DVB-H[2] as well as Digital Multimedia Broadcasting (DMB)[3] for hand-held devices.

On-demand Streaming Applications On-demand applications deliver pre-encoded content individually to large numbers of receivers at request. While users expect high resolutions and tolerate limited quality degradation only, they accept startup delays and rebuffering. For scalability reasons the topologies are based on multi-

[1]`http://www.dvb.org/`
[2]`http://www.dvb-h.org/`
[3]`http://webapp.etsi.org/action/PU/20050628/ts_102428v010101p.pdf`

ple streaming servers and streaming proxies [30], where a single receiver is served by a single unicast or multicast connection. An alternative approach are peer-to-peer overlay networks where each receiver is provided with short stream sections from multiple peers [31, 32].

Channel characteristics are often unknown because on-demand streams are usually delivered over shared best-effort packet networks like the Internet. Although back-channels are available, scalability requirements limit the amount of receiver feedback. Error-control schemes such as forward error protection, retransmission and hybrid approaches are feasible, but they depend on scalability requirements of the stream sender. In order to serve receivers with diverse device capabilities, scalable encoding may be employed to scale down resolution and decoding complexity of a stream.

Examples of on-demand streaming systems range from Video Blogs and community platforms like youTube to on-line TV recorders, education platforms, commercial-grade video-on-demand services for movies and Digital Cinema systems.

Editing and Composing Applications Editing and composing systems are used in TV and video-studio environments, newscast production and movie post-production environments. Common tasks are transcoding, effect rendering and re-encoding, whereas stream formats are typically loss-less. Streams are either pre-encoded or live encoded by camera hardware. As such systems require quick and random access to high-quality streams for searching and editing, the focus lies on reliability, quality and scalability to large data streams rather than on many users and many concurrent streams. Although the common access patterns require low startup delays, the end-to-end delay is of secondary interest. Hence, networks, storage systems and workstations are well provisioned for high-throughput and dedicated exclusively to a single application. Because network errors are unlikely and rebuffering is allowed, retransmission schemes are adequate. Application topologies are fixed to a small set of storage systems and streaming servers and a moderate number of workstations.

Conversational Applications Conversational applications interactively exchange voice and video streams for personal communications and multi-party conferences. The content is live encoded under strict

Application Class	**Conversational**	**Live Art**
E2E Delay	very low	very low
Startup Delay	low	low
Rebuffering	no	no
Resolution	low	high
Coding	efficient, loss-resilient, low-complexity decoder	efficient, loss-resilient, decoder
Error Control	FEC	FEC
Channel	uncontrolled shared	uncontrolled exclusive/shared
Topology	uni/multicast back-channel	unicast back-channel

Tab. 2.2 : *Requirements of different streaming application classes (continued).*

real-time constraints ($delay < 100ms$), which heavily restricts error-control options and playout-buffering at receivers. Hence, conversational applications are sensitive to jitter. Therefore forward error protection techniques are preferred. Low resolution and short time quality degradation is acceptable in some applications. The stream pattern often divides into talkspurts and periods of silence which may be exploited for error control and resynchronisation. Due to limited device capabilities, encoder and decoder complexity must be low.

Application topologies of multi-party conferences either consist of direct point-to-point connections or multiple point-to-multipoint connections. Network handover procedures are common in mobile wireless communication systems. When a back-channel exists, it may be used for feedback to make the sender aware of the forward-channel quality experienced by each receiver. Although feedback has limited utility for retransmission schemes due to short delays, it can be used for channel estimations to protect future data more efficiently and for encoder control to limit loss propagation by adaptive refresh schemes.

Examples of conversational applications are personal videophones and interactive games, business-quality conferencing and telepresence systems, highly reliable systems for rescue services as well as smart rooms and augmented reality systems for research, engineering and construction.

Multi-Party Live Performance Applications Live performance applications virtually interconnect and assist artists, such as musicians, video artists and dancers during performances, rehearsals and recording sessions. The multimedia contents is either captured and encoded live (e.g. sensor input, video, audio, and MIDI signals) or pre-encoded and stored off-line (e.g. video sequences, sound samples, pictures, and 3D object models). The performance requirements are very strict. For interactivity, musicians require ultra-low delays ($< 20ms$) and exact synchronous delivery of events from multiple sources. Loss or quality degradation may be intolerable. Visual art may tolerate larger delays especially when output is rendered frame-wise (e.g. every 40 ms), while distortion due to loss may be distracting. Although low delays restrict error control to forward correction schemes, receiver feedback can be used to tailor the amount of redundancy to observed channel conditions. Local studio or stage networks are usually exclusively available to a single application, but when members are connected via Internet paths, they experience bandwidth and delay variability.

In general, streaming applications are delay-sensitive, bandwidth-intense, and partially loss-tolerant. There are, however, differences between application classes with diverse requirements that can considerably affect the overall system design. Real-time constraints differ between application classes. While conversational systems have very tight constraints ($< 20-200ms$), live broadcasting is less severe affected by delays ($< 500ms-2s$) and on-demand applications have even looser constraints ($> 500ms-2s$) [26,27]. Live-encoding requires real-time processing and may be less efficient than offline-encoding which can employ an additional analysing stage for pre-processing.

Channels can be static with fixed or guaranteed bandwidth, delays and loss, or dynamic where these properties vary over time. In general, pre-encoded streams are harder to adapt to variable channel conditions than live encoded streams, but scalable video coding schemes [33] try to close the gap. For low loss rates it is important to match stream bitrate to channel bandwidth. For static channels, constant bitrate (CBR) streams are more appropriate although the encoding quality may vary due to variable complexity of sequences. Variable bitrate streams (VBR) in combination with adaptive rate-control schemes (see section 2.3.4) are preferred for dynamic channels and for high-quality applications. The sender-side knowledge about the channel may differ between application classes, depending on back-channel availability and scalability constraints. With back-channel and receiver feedback, a sender can adapt a stream to estimated channel condition, while without feedback streams must be prepared for worst-case conditions.

This taxonomy is intended to show the diversity of available streaming systems. It is not supposed to be complete and exclusive in any sense. Some applications may not fit into a single class, for example Internet-based radio broadcasting stations. Although they clearly have broadcasting character, they also share some attributes with on-demand systems, when receivers request a stream.

2.1.2 Application Architectures and Topologies

Designing useful middleware interfaces heavily depends on knowledge about how streams are generated, processed and consumed by applications and how streaming mechanisms integrate into application architectures. Hence we first discuss common components found in virtually any streaming application. We then show how encoders are embedded into distributed architectures and which protocol modules and wait queues are placed along a stream's data path. Finally, we present some common communication topologies.

Application Components The general architecture of any streaming applications resembles the pipe-and-filter pattern [17, 34, 35], composed of multiple processing stages, interconnected by abstract communication channels. In combination with stream splitters and stream multiplexers, programmers can build complex *filter graphs*.

Single filter stages are self-containing entities with a clearly defined interface, while communication channels loosely couple filters by explicit data paths. Filters contain, for example, stream encoders and decoders, mixer, filter, I/O and display components. Filters are not required to be aware of the total graph, they must only know their direct neighbours or the channels which connect them. This decouples topology and communication aspects from filter implementations and reduces application complexity. An application can compose the filter graph from simple building blocks, and configure, control and monitor its operation. Programmers may use special execution models to define which filters may run concurrently and which filters must run sequentially [17, 36]. By a clever combination of communication transparency, anonymity between filters and encapsulation of complex functionality, the pipe-and-filer pattern is thus inherently scalable and adaptable.

The following architectural components and data-flow entities are common to distributed streaming applications:

Filters represent the processing nodes of a filter graph. They are application-level entities which generate, process or consume one or multiple media streams. Hence, there are three types of filters, *stream producers*, *stream filters* and *stream consumers*. While a producer initially generates or imports a stream in a specific format from external devices such as cameras or storage media, a consumer finally exports a stream for display, storage or further external processing. Stream filters actually manipulate streams, either by changing the encoding format, or by changing the content. Filter implementations may use a private thread or may share threads from a common pool, but they may also be passive entities that are called from application threads.

Bindings [37] represent directed edges of a filter graph, used to pass stream data between adjacent filters, either in a point-to-point or point-to-multipoint topology. Bindings abstract from location, communication failures and the actual implementation details of transport protocols. They usually have a configuration interface to configure communication properties and sometimes a monitoring interface to unveil lower-layer transport channel characteristics. Bindings can define semantics that identify which operations are available and which communication guarantees the binding is willing to give. Bindings are usually passive, thus they require an external thread for execution.

Ports [34] represent typed input and output interfaces of filters which are used by the filter as communication end-points for receiving and sending stream data. Filters can have multiple input and output ports. Ports ensure the type-safety of streams by validating the type of opposite ports when connections are established. Note that ports and bindings are alternative concepts to interconnect stream processing stages. Like bindings, ports can hide protocol details and allow for the inspection and control of communication-related aspects such as channel state, flow-control and error-control. The difference is that bindings can recursively contain graphs of filters and other bindings, while the external interfaces of the contained graph are exported as the interface of the new binding. This allows for greater flexibility in composing applications, but effectively prevents network-adaptive applications from monitoring channel conditions when hidden by a binding

Streams represent sequences of semantically related and typed data units that flow through a potentially distributed filter graph. A single stream originates from one producer and ends at one or multiple consumers or multiplexers. At its path through the filter graph, a stream may pass multiple filters, may be transcoded, multiplexed and split. When two streams are multiplexed, the semantical relationship between the data units in each original stream are enhanced by a new relationship to data units in other streams. Therefore, incoming streams loose their identity and end at a multiplexer while a new stream with new semantics and a new identity originates there.

Flows represent sections of a stream which pass through a particular port or a particular binding. A flow begins at the output port of a filter and ends at the input port of a second filter. Hence, in multicast settings there exist multiple flows. At the first glance, flows seem similar to streams, but there are important differences: Streams describe high-level information flows through the application, whereas flows describe low-level data flows between application modules. While streams are defined from the global perspective of an application, a flow is defined from the local perspective of a single port or a single binding. Hence flows share content attributes with the stream (e.g. creation time, author and location), but they may have specific formats and bitstream layouts. Two flows of the same scalable bitstream, for example, may represent a similar content but at different operation points and thus different qualities.

We will later use and extend the abstractions of ports and bindings to design our middleware interfaces. The definitions of streams and flows currently have no counterpart in one of our middleware abstractions.

They are provided for completeness and as references for future extensions, for example as an abstraction for global resource management or security and privacy concepts.

Encoders in a System Model When media encoders and decoders are integrated into the architecture of a streaming system their control and data paths must be carefully designed to avoid extra delay. Figure 2.1 shows the system model of streaming applications from a coder perspective [38]. At the encoder side, a media signal is first compressed by a source coder to reduce redundancy and irrelevance. Often, this step is loss-less to achieve high coding efficiency. Then, a channel encoder is used to prepare the compressed media stream for delivery of a communication channel. When information about the channel is available, such as available bandwidth and error patterns, source and channel coding is adapted to ensure optimal delivery. Note that this general model does not assume special system architectures or special software layers where channel coding is performed. The channel coder may be a transport protocol above a general-purpose packet-network stack or a special-purpose link-layer in an embedded device (e.g. a GSM phone). Channel encoding may involve packetisation, error protection, modulation and transport level flow control.

At the decoder side, the reverse operations are performed to reconstruct the output signal. A channel decoder extracts data from the channel, conceals errors and forwards the stream to the source decoder. Feedback from both, the channel decoder and the source decoder may be sent to the encoder side to adapt the encoding process to the observed

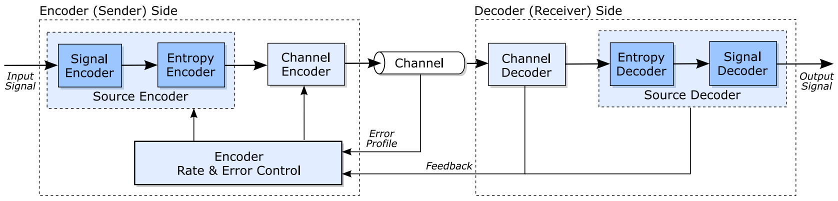


Fig. 2.1 : *Video coding system model. Encoder and decoder contain separate blocks for signal and channel coding that may be coordinated. When feedback is available, the encoding process can even be adapted to channel and decoder state.*

channel characteristics and the decoder state. The communication channel models an arbitrary digital delivery facility, such as for example a DVD media, a circuit-switched or a packet-switched network.

Without information about channel conditions and decoder state signal encoding must rely on predictions. Even with feedback, the feedback contains observations about the past channel behaviour only. For fast fading wireless channels predictions usually become wrong. Depending on the applications scenario and available information about the channel different error protection techniques can be employed (see section 2.3).

Data Paths and Transmission Delays Channel encoders or transport protocols perform complex tasks to ensure timely delivery and protect streams from uncontrolled channel errors. Decoders require in-time data and low error rates to reconstruct a reasonable signal quality [39]. Figure 2.2 displays typical modules and information flows in media streaming transport protocols, and shows locations (as shaded boxes) where delay is added to stream forwarding.

At the sender side, a streaming protocol may classify and partition data units by importance to enable unequal error protection. When necessary, the protocol also fragments data units to meet the maximum transmission unit size (MTU) of the network path. When forward error correction (FEC) techniques are used (see also section 2.3.3), they may add a packetisation delay [40]. Rate control, congestion control and packet scheduling schemes (see section 2.3.4) add an additional delay because they smooth traffic in order to avoid congestion-related loss. Next, the network may introduce a variable transmission delay which

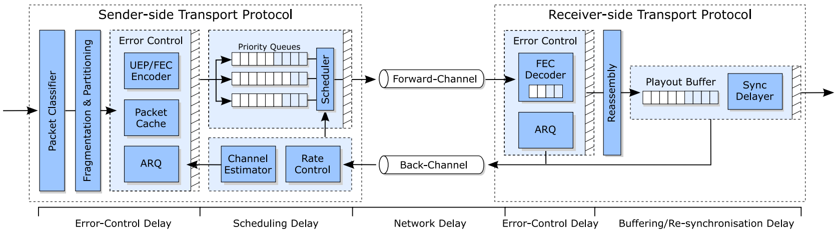


Fig. 2.2 : *Data-paths and delays in streaming transport protocols. The shaded boxes indicate potential sources of delay.*

depends on back-off delays after collisions in multi-access networks, link-layer retransmission schemes and queuing delays in packet routers [23]. At the receiver, FEC reconstruction, de-interleaving and reassembly of fragmented data units must wait until all fragments of a particular FEC block or data unit are received, while a playout buffer additionally delays a stream to compensate for network jitter, retransmission times and reordering. Some transport protocols and scheduling schemes operate on data units in a sliding transmission window [1]. Such protocols require an additional initial *pre-roll delay* to fill the transmission window.

Application Topologies Streaming applications may use a variety of communication topologies to exchange media streams between distributed processing stages. The selection of a particular application topology and its implementation on top of a particular network topology impose constraints on scalability and receiver feedback.

Besides simple point-to-point and multicast topologies there are many options to distribute streams in a continuous, reliable and scalable way to one or many receivers either with or without feedback [41]. Hierarchies of dedicated relay nodes can increase scalability and fault tolerance of large-scale content-delivery networks by balancing load [30]. Application-level multicast trees established in overlay networks [31, 32] may also achieve high scalability and fault tolerance while load is equally distributed across multiple participants and network links. Multi-homed devices and multi-path streaming are known to increase communication quality and reliability for end-nodes connected to wireless networks [42], while soft-handover and roaming in mobile scenarios can be achieved by streaming proxies [43].

2.2 Streaming Protocol Standards

Several protocol standards for transporting data streams and stream-related control traffic over packet networks are defined by different standardisation bodies. This section gives a brief overview of available protocols and their functionality.

2.2.1 MPEG Transport Specifications

Todays commercially successful audio and video coding standards, such as for example MPEG-1/2/4 are defined by the Moving Picture Experts

Group (MPEG)[4], a working group of the International Organisation for Standardisation (ISO). The new MPEG-4/H.264 Advanced Video Coding (AVC) and Scalable Video Coding (SVC) standards are defined in cooperation between MPEG and the International Telecommunication Union (ITU). Besides bitstream syntax, bitstream semantics and the decoding process, a part of each coding standard is concerned with the delivery of encoded streams over various transport technologies, such as data storage (Digital Versatile Disc, DVD), digital television systems (DVB), or packet networks. Because different encoding standards target different application domains, the transport-related features may considerably differ. MPEG-2, for example, was developed for broadcast-quality television services, reliable high-performance networks (Asynchronous Transfer Mode, ATM) and optical storage (DVD), whereas H.264/MPEG-4 AVC specifically targets packet networks.

MPEG-2 defines two container formats, *Transport Streams* for delivery over network channels and *Program Streams* for storage media such as discs. MPEG-2 transport streams define schemes for multiplexing multiple elementary streams, but no error-control. The standard assumes error protection and isochronous delivery from the underlying network. For compatibility with ATM networks, transport stream packets are 188 byte long (4 times the payload size of ATM cells). Currently, MPEG-2 transport streams are used for digital television broadcast over cable, satellite, and terrestrial paths and high-quality video conferencing systems.

The flexibility of MPEG-4 [44] required a new transport framework, the Delivery Multimedia Integration Framework (DMIF) [45]. The purpose of DMIF is to hide transport details from encoders and leverage interoperability between different implementations. Because MPEG-4 defines a rich set of different media types, such as audio, video, objects, shapes, etc., which can be dynamically composed into sequence, DMIF provides functions to multiplex and synchronise streams of different type and methods to create and release channels dynamically. Internally, DMIF uses stream and transport multiplexer stages, defines a generic channel establishment procedures as well as a generic Quality-of-Service descriptor to specify quality constraints for different transport layers. Besides transport streams and DMIF, implementations and standards for mapping MPEG-2 and MPEG-4 streams to the Internet streaming protocol RTP do exist [46–48] (see also the next section).

[4]`http://www.chiariglione.org/mpeg/`

The H.264/MPEG-4 AVC standard was specifically designed for low-complexity delivery over packet networks [27,49]. H.264 defines a Video Coding Layer (VCL) for signal compression issues and a Network Adaptation Layer (NAL) for encapsulating encoded data into network packets, MPEG-2 transport stream packets or file formats. The NAL ensures that when a packet containing a NAL unit is lost, the VCL regains synchronisation to the bitstream at the next successfully received NAL unit. This effectively avoids an extra synchronisation layer and special resynchronisation markers in the bitstream, but requires transport layers to mask bit-errors as packet loss. The low complexity of H.264 NAL is also expressed in the reuse of the one-byte NAL unit header as the payload header for the H.264 RTP payload format [50]. The H.264 scalability extension H.264/SVC [51,52] enhances the NAL unit concept by extra header fields to enable low-complexity in-network scaling.

2.2.2 Multimedia Protocols for the Internet

Many protocols for multimedia streaming applications have been published in the literature. While protocols from the research community usually focus on specific features only, standard protocols aim to provide complete and interoperable solutions. In this section we briefly discuss the main multimedia transport and signalling protocols for the Internet, published by the Internet Engineering Task Force (IETF). IETF Protocol standards as well as the MPEG/ITU encoding standards are adopted by other standardisation bodies which specify particular application environments, such as the Internet Streaming Media Alliance (ISMA)[5].

Transport Protocols The real-time transport protocol (RTP) [5] is an extensible protocol framework for end-to-end delivery of real-time data streams. RTP consists of two closely-related parts, the real-time transport protocol (RTP), to carry data that has real-time properties and the RTP control protocol (RTCP) to allow scalable receiver feedback and monitoring of data delivery. RTCP is only intended to carry a limited amount of feedback and control information. The majority of control data should be exchanged via other signalling protocols, described later.

A key goal of RTP is to provide a very thin transport layer without overly restricting the application designer. RTP supports payload type identification, sequence numbering, timestamping and even multicasting

[5]`http://www.isma.tv/`

features if provided by the underlying network, but it lacks reliability and quality-of-service guarantees. RTP itself does not ensure timely delivery, does not prevent out-of-order delivery and does not guarantee delivery at all. Although RTP is independent from the underlying transport, it is usually layered on top of UDP [53]. Applications can and must arrange for compensation of delays and loss themselves. RTP does, however, define payload format specifications which describe how a particular format, such as encoded audio or video, is mapped to RTP packets to increase robustness. Mappings follow the principles of application level framing (ALF) [54] while RTP profiles allow for the inclusion of additional header fields and semantics. Thus RTP facilitates joint source-channel coding for Internet channels [55]. RTP is implemented as a set of functions or as a library which is directly merged into application processing, rather than being a separate transport protocol layer. The restricted features of RTP are repeatedly criticised [56]. RTP provides no real-time support as its name suggests, lacks congestion-control, heavily restricts control traffic and uses many UDP ports, at least at gateways. Although the design decisions are sound from the perspective of scalability and recent extensions are made to alleviate the problems, many applications require richer communication support.

Promising solutions to RTP's lack of congestion control are to use the Datagram Congestion Control Protocol (DCCP) [8, 57, 58] as transport instead of UDP, or the direct combination of RTP with TCP-friendly rate control (TFRC) [59–61]. In addition to the unreliable unicast service of UDP, DCCP provides connection-oriented, congestion-controlled transport. It allows to select TFRC and TCP congestion control styles, supports path MTU discovery and integrates smoother with firewall schemes.

While traditional Internet protocols support either full reliability or no reliability, the Stream Control Transmission Protocol (SCTP) [62,63] supports a partially reliable mode SCTP-PR [64], where applications can choose reliability on a per-message basis. For example, SCTP-PR allows to specify a deadline for the protocol to indicate when to give up sending retransmissions. Besides this, SCTP is a connection-oriented, congestion controlled protocol like TCP, supporting multi-homing, failover and multiple streams per connection. Experiments on multimedia streaming with SCTP-PR are documented in [65, 66].

Signalling Protocols The IETF defines the Session Description Protocol (SDP) [7], the Session Initiation Protocol (SIP) [6, 67] and the real-time streaming protocol (RTSP) [68] as the principal means to establish and control streaming sessions over the Internet. SDP is a general-purpose format for describing session parameters such as names, times, media formats, transport protocols and address information for session establishment, whereas negotiation of session content or media encoding are outside the scope of the protocol. SDP can be embedded into several transports, including SIP, the Session Announcement Protocol (SAP) [69], RTSP, or electronic mail using the MIME extension. SIP is intended to initiate and reconfigure multi-party streaming sessions, such as Internet telephone calls and conferences as well as sessions with gateways to Public Switched Telephone Networks (PSTN). SIP provides means to locate nomadic users, negotiate compatible media types, and authenticate and authorise users. RTSP can establish and control the on-demand delivery of one or multiple media streams, while the actual stream data is delivered by other protocols such as RTP. For example, RTSP allows a streaming client as well as a streaming server to describe streams (e.g. in SDP), request and pause stream delivery and invite other participants.

Other signalling standards originating from the telecommunications industry are H.323, defined by the Telecommunication Sector of the ITU (ITU-T). Although H.323 is similar to SIP, it was available earlier. Today, H.323 is widely deployed in Voice over IP services and Internet videoconferencing systems. Because it provides signalling interoperability with digital telephone networks, it is most popular in this sector.

2.2.3 Streaming in Mobile and Wireless Networks

Standards for mobile packet networks are mostly defined by industry-oriented standardisation bodies to increase the interoperability between device manufacturers and network operators. The 3rd Generation Partnership Project (3GPP)[6] specifies streaming and interactive multimedia services for mobile cellular telephone networks, while the Open Mobile Alliance (OMA)[7] claims responsible for mobile broadcasting over DVB-H and 3GPP. In order to increase the interoperability with existing Internet services, these standards mostly follow the IETF proposals, but usually restrict their flexibility to decrease implementation complexity.

[6] `http://www.3gpp.org/`

[7] `http://www.openmobilealliance.org/`

Most of the protocols in wireless packet networks are concerned with link-layer connectivity, efficient link-layer error control schemes and fair resource sharing, rather than transport protocol issues. The intended transport protocol for multimedia traffic in 3GPP networks is RTP over IP. Hence, the transport-level challenges are similar to wireless local-area networks.

2.3 Network-adaptive Multimedia Streaming

Delivering time-sensitive media streams over unreliable packet networks is challenging since neither resource availability is guaranteed nor performance characteristics in terms of delay or loss are predictable. Hence, applications are required to adapt to changing network conditions. Recent advances in network-adaptive media streaming indicate that quality and error resilience can be significantly improved by robust bitstream packetisation, adaptive encoding schemes and scalable encoding at the encoder level. At the transport level adaptive error control, adaptive rate control and buffer management have proved to be proper solutions.

In this section we first discuss general sources and effects of network errors. After that we present mechanisms proposed for the signal coding layer and for the transport layer to alleviate the problems of packet networks.

2.3.1 Sources and Effects of Network Errors

Transmission over best-effort packet networks like the Internet exposes media streams to several classes of errors, such as potential deadline violations, loss, and reordering. Most of the error classes are related, either because they share the same reason or because a technique to overcome one error is likely to introduce another error. The following three main classes exist:

Timing Errors Transport delay and periodic timing of multimedia streams are influenced by several effects in packet networks. While propagation delays of links are constant, the best-effort store-and-forward routing disciplines of packet networks and the back-off schemes of shared links can introduce a variable end-to-end delay which depends on the actual network load. Additionally, reordering and duplications are introduced by multipath routing and timeout-based link-layer retransmission schemes.

When packets arrive too late, they often become useless because the receiver must decode and render the contained signal (audio samples or video frames) at a constant rate. Hence, deadline violations effectively translate into loss errors. When data is unavailable for decoding, a decoder can try to conceal the missing information [70] from previous stream data, but often the signal is severely distorted. Predictive encoding schemes, such as MPEG video codecs, can especially suffer from loss because distortion in reference frames can propagate to future predicted frames until the next intra-coded frame is received. Although it may be possible to use late packets to repair reference frames [71], most decoder assume in-order delivery of bitstreams.

Timing errors are typically addressed by introducing an extra buffering delay at the receiver to wait for late and reordered packets. Buffering is, however, strictly limited in conversational and interactive applications because they require very low delays. On-demand and broadcasting applications can, in contrast, benefit from large jitter compensation buffers due to their relaxed delay requirements although buffer space may become a limiting factor.

Bit-Errors and Packet Loss Different types of loss errors occur in different network environments. While congestion is the dominant source of loss in fixed local and wide-area networks, bit-errors are dominant in wireless channels. Congestion occurs when a router at a bottleneck-link observes high network load so that incoming traffic does no longer fit into the router queues and packets must be dropped. For end-nodes this drop happens uncontrolled because neither the time nor the amount of dropped packets can be estimated in advance. In contrast to traditional drop-tail policies, more Internet routers become equipped with sophisticated drop policies such as random early detection (RED) [72]. Such policies provide earlier indications to end-to-end transport protocols so that they can throttle their output rate to avoid congestion.

The link-layers of packet networks usually translate bit-errors into packet loss because it is unknown if bit-errors destroyed parts of an upper-layer protocol header or the application payload. Although multimedia applications are tolerant to some bit-errors, protocol implementations are not. When corrupted packets are passed to upper-later protocols it is likely that wrong routing and multiplexing decisions are made if packet header information was corrupted by the bit-error.

There are two prominent approaches for error control at the transport level [40], retransmissions and forward error correction (see section 2.3.3 for more details). Although both can compensate for isolated and burst loss, they have the potential to increase congestion because both increase the bitrate of the stream. Moreover, retransmitted packets must arrive before their deadline and the extra packets should not violate deadlines of subsequent data.

Bandwidth Limitations Although bandwidth limitations also result in loss errors, we treat them separately because in contrast to network packet loss, this loss may be controlled by transport protocols. Available bandwidth is generally unknown and time-varying in the Internet. It depends on network load imposed by concurrent data streams and unpredictable conditions of wireless links that switch between modulation schemes when signal quality changes. Mobile users that roam through different networks are even more affected because they may approach a heavily loaded base-station or a lower-capacity network after handover. Limited bandwidth may be known to a transport protocol, either through monitoring a local link-layer or by using a bandwidth estimation protocol. Instead of waiting until the network randomly drop packets, the stream sender can selectively discard packets which are old or less important for signal reconstruction in favour of more important ones. While rate-control and congestion-control reduce the probability of loss, they cannot guarantee loss-less delivery. Even with rate-control, streams may experience loss in short-time periods of congestion which are shorter than a round-trip-time, and especially when competing traffic is not rate-controlled.

2.3.2 Error-Resilient Signal Encoding

Recent research on encoding standards has largely focused on efficiency and error-resilience concepts, especially targeted at error-prone wireless and packet-based delivery channels [3, 21, 73, 74]. Error resilience techniques at the signal encoding level divide into the three classes: (1) forward, (2) interactive and (3) concealment techniques. They are applicable at different encoding stages and in different environments depending on available resources and back channels. We omit concealment techniques because they are less relevant for communication engineering. A survey on concealment approaches can be found in [70].

Forward Techniques Forward techniques are employed by encoders in order to proactively protect a bitstream against errors. They work by adding a controlled amount of redundancy to the encoded bitstream either at the entropy encoder or the signal encoder stage. Common techniques in MPEG-4 [44] are for example:

Forward Error Correction (FEC) at the encoder level adds redundancy (e.g. parity bits) to the bitstream after entropy coding in order to detect and potentially correct bit errors. This is only necessary if the transmission channel lacks bit-level error control.

Resynchronisation Markers are uniquely decodable binary codes inserted in the bitstream. After losing synchronisation, a decoder may resume correct decoding upon reaching such a point. This limits error propagation by localising the effect of an error to the area between two resynchronisation markers. If used too often, they create overheads, but limit the damaged area more tightly. The distance between markers may either be variable or fixed. If the distance is related to the network packet size, delivery becomes more robust against packet loss in the Internet.

Reversible VLC (RVLC) is an error resilient entropy coding method which reduces the damaged area after a bit error by creating a bitstream which is decodable in forward and backward direction. The decoder stops at detected bit errors, forwards to the next resynchronisation marker and decodes backwards from this marker. RVLC algorithms must produce symmetric code words and it is required that invalid code words are detectable. If bit errors produce valid code words, the error may be detectable during signal reconstruction.

Data Partitioning rearranges encoded data of a single structural element (e.g. a frame or a slice) into groups according to their sensitivity to errors. Partitions are transmitted in importance order and the last partition may be followed by a resynchronisation marker. If a random bit error occurs and subsequent data is lost, it is more likely that important data is still valid. Since important data comprises a small fraction of a bitstream only, it can be stronger protected using RVLC and FEC without a severe loss in coding efficiency. In packet-switched channels each partition may be placed in a separate packet to confine the impact of packet loss.

Interleaving is used to overcome burst errors by separating neighbouring information as far as possible. Interleaving may be used in signal compression to spread pixels or macroblocks and in entropy coding to spread encoded bits or symbols. Because burst errors affect interleaved streams in non-continuous regions of a frame only, error concealment schemes may become more effective.

Forced Redundancy intentionally keeps or adds redundancy during signal compression instead of removing all statistical and psychovisual redundancy. This restricts spatial and temporal error propagation, but may significantly decrease compression performance. Examples are periodically encoded intra-mode macroblocks, repeated header information and motion vectors and even redundant frames.

These techniques are not mutually exclusive. They may be combined to construct complementary solutions. In general, their effectiveness increases when decoder feedback is available. There is always a trade-off between bandwidth consumption and error robustness.

Interactive Techniques If a back-channel with acceptable latency exists and the stream is encoded live, decoder and encoder can cooperate to minimise the effects of transmission errors. This works either by adapting the encoding process or by retransmitting corrupted data. Interactive error-resilience is typically reactive because the encoder adjusts its operation when feedback arrives. Feedback can either be negative or positive, explicitly informing the encoder of errors or acknowledge the correct reception of stream data. Encoder-level interactive techniques include:

Adaptive Reference Selection informs the encoder to avoid using specific frames as references because they are corrupted at the decoder. This requires short round-trip times to become effective and extra reference picture buffers at the encoder and the decoder to store multiple reference candidates.

Adaptive Intra-Refresh includes intra-coded data on request to effectively stop error propagation at the receiver. Alternatively it adapts the amount of intra-refresh to quality reports sent by the receiver.

Error Tracking reconstructs the error propagation process of the decoder from feedback about lost data units. Hence, the encoder must only update the distorted regions in new frames by intra-coding. Although error tracking adds no additional delay and requires no extra storage at the decoder, it increases encoder complexity.

Scalable Video Coding To overcome limitations for multicast and multi-rate delivery, scalable bitstreams and appropriate encoding concepts have been developed recently [33, 51, 75–77]. Scalable coding techniques split the compressed signal into separate embedded bitstreams, where each bitstream contains a special fraction of the encoded signal only. Scalable streams can be decoded at different bitrates with a graceful degradation in quality. This comes at the cost of higher encoding complexity and reduced compression efficiency. For video streams, scalability is performed in spatial, temporal and signal-to-noise (SNR) dimensions. Spatial scalability increases the resolution of a single picture or a picture element. Temporal scalability increases the frame rate of a sequence by using one or more layers of discardable bi-directionally coded frames (B-frames). SNR scalability increases the signal fidelity of a frame without increasing the spatio-temporal resolution. This is achieved by varying the quantiser step size and the number of encoded coefficients.

Scalable streams benefit from increased robustness to bandwidth fluctuations. They can be adapted to heterogeneous and varying receiver and channel characteristics at low complexity. Streaming servers can serve diverse receivers and even heterogeneous multicast groups from one scalable pre-encoded bitstream while transport protocols can unequally protect different layers according to their importance.

Several methods for generating scalable bitstreams do exist. Some methods define a hierarchical order between quality levels, where most important information is contained in a *base layer* and less important data in one or more *enhancement layers*. The base layer is essential to reproduce a minimum quality signal while enhancement layers extend the signal quality if available. Other techniques distribute the data more equally across layers, so that any combination of layers is usable for signal reconstruction. Published concepts include:

Layered Coding [55] splits a bitstream into at least two separate parts: a base layer and one or more enhancement layers. While the base layer provides moderate quality, adding enhancement layers improves quality. The decoder requires at least the base-layer to reconstruct a signal at all.

Multiple Description Coding (MDC) [77] distributes the information uniformly across equally important layers to circumvent the problem of corrupted base layer data. A single description suffices to reconstruct the signal in a basic quality and every additional description improves the quality. MDC achieves best results if descriptions are delivered via independent channels with unrelated error probability.

Fine Granularity Scalability (FGS) [76,78] operates at the entropy coding stage and applies bit-plane coding instead of VLC coding. FGS uses base and enhancement layers, but unlike layered coding, a partially received enhancement layer contributes to signal reconstruction. Hence, quality degrades gradually when error rate increases.

Sub-band and Wavelet Coding Although 3D sub-band coding techniques and the more recent wavelet coding schemes [38, 79] are considered to have poor coding efficiency compared to motion-compensation schemes, they may exhibit better error-resilience characteristics because no long-range dependencies between frames are introduced [80]. Due to the limited use of these schemes in current video coding standards we do not consider them in this work. We note, however, that the content-aware interfaces and communication architectures developed in this thesis are in particular applicable to wavelet encoding schemes because different bit stream segments have unequal significance across temporal sub-bands [79].

2.3.3 Adaptive Error Control

At the channel coding level, applications have several options to deal with network errors, depending on the availability of time, buffer space and back-channels. Applications can re-send missing data, either stored in a sender-side buffer or re-generated on request. An application can also add redundant packets to a stream or re-order packets to avoid consecutive packet erasure during loss bursts. While there are good surveys on this topic in [26] and [40], we will briefly summarise existing error-control options in the following.

Robust Bitstream Packetisation One of the key concepts to achieve reasonable error resilience in multimedia communications over packet networks is the Application Level Framing (ALF) principle [54]. ALF assumes that the application defines the unit of transport as Application Data Unit (ADU) itself. Because a receiver can immediately process successfully delivered ADUs, the application can effectively cope with misordered or lost data. Hence ADUs become the minimum unit of error recovery. When ADUs equal network packet, a loss can only destroy complete ADUs, and all packets that are received are known to contain data that can be processed independently of other packets.

RTP, for example, follows the ALF principle and defines packetisation rules [81] for mapping encoded bitstreams to packets that allow to identify and decode a packet irrespective of whether it was received in order or whether preceding packets have been lost. RTP suggests that packet boundaries should match bitstream element boundaries and that a codec's minimum data unit should never cross packet boundaries. In network-adaptive encoders such as H.264, this concept even influenced the design of the coding layer.

Forward Error Correction (FEC) FEC schemes systematically add redundancy to a packetised media stream to enable the detection and correction of bit-errors and the reconstruction of lost packets at the receiver. Block-based FEC schemes apply a systematic block-code (e.g. Reed-Solomon or Tornado codes [40, 82]) to a group of k consecutive packets to obtain a block of n packets ($n > k$), whereas all n packets contain modified payload. A receiver is able to reconstruct the k original packets when it receives at least any k FEC packets of the group. For a large k, block-based schemes can increase the overall end-to-end delay because a sender must wait until k original packets are available while the receiver must wait until at least k FEC packets are received. Some FEC schemes can even become ineffective when loss bursts are larger that $(n - k)$ because then no data can be recovered at all. Because FEC only adds a low delay (for small values of k) and does not require a back-channel it is attractive for conversational and multicast/broadcast applications. Simple FEC techniques, however, introduce a constant overhead that reduces throughput when the channel is relatively error-free.

Unequal Error Protection (UEP) In order to increase the efficiency of FEC-based schemes, several unequal protection schemes have been proposed [83–86]. Unequal error protection assigns different amounts of redundancy to bitstream sections of different importance in order to provide different levels of error recovery. UEP schemes may, for example, distinguish between different frame types [85], bit-planes and layers of scalable streams [84] and multiple descriptions [86]. Finding the optimal amount of redundancy is usually formulated as an optimisation problem which considers the current channel conditions (loss rate, loss pattern and available bandwidth), the importance of bitstream sections and the dependency between bitstream sections. For reasonable performance, UEP schemes require perfect knowledge about the channel conditions or at least good estimations, which may be unavailable in fast fading wireless channels. All schemes proposed so far consider a special encoding format or bitstream structure. A general model that addresses the importance and dependency semantics across encoder families is still missing.

Automatic Repeat Request (ARQ) ARQ schemes use retransmissions to recover from lost or corrupted packets based on positive or negative receiver feedback [87, 88]. Although ARQ schemes are considered inappropriate for interactive applications because they add a considerable delay (at least one round-trip time), they are a simple, efficient and flexible solution for other streaming scenarios. Retransmission requires sufficient buffer space at the sender to store stream data and at the receiver to compensate for jitter introduced by the retransmit delay.

Re-sent data can either be the original version or completely new version that supersedes the old. The sender may even choose to avoid sending a retransmission when data is less important or a deadline has passed. Because retransmissions are widely used at wireless link-layers, content-awareness becomes especially important to ensure real-time delivery. Multiple selective or content-aware ARQ schemes have been studied in [89–92].

Hybrid Error-Control Schemes Hybrid schemes combine the benefits of multiple error-control techniques to increase either their efficiency or their robustness to certain loss patterns. Recall that the main problems of FEC were the fixed amount of redundancy and the sensitivity to burst loss. Combined with ARQ, two general types of packet-level hybrid FEC/ARQ exist [26]. The first uses FEC data in every transmitted packet and retransmits a subset of the same packets when the number of received packets is insufficient for reconstruction. The second hybrid FEC/ARQ scheme adds no redundancy to the first transmission, but sends redundant FEC packets in retransmissions instead of original data. While the first scheme is attractive for single receivers and conversational applications, the latter has advantages for multicast groups where some receivers observe shared loss while others observe individual loss in a group of packets. A single FEC packet is sufficient to let all multicast receivers recover from an arbitrary lost packet per group. Adaptive FEC/ARQ schemes [93–96] that consider channel state may even further increase the effective throughput. Other hybrid schemes combine FEC with packet interleaving [97, 98] to overcome burst losses, and adaptive FEC with selective drop [99] to optimally exploit available bandwidth.

Packet Scheduling Although packet scheduling is complementary to error-control schemes, the selection of a particular delivery schedule may heavily influence the efficacy of error-control. Scheduling policies decide whether, which and when to transmit or retransmit data packets over a bandwidth-limited network connection.

A scheduler may choose to reorder packets to gain more time for potential retransmission [1, 100, 101], scramble or interleave packets to improve resilience against burst loss [102, 103] and drop packets after their deadlines or when queuing capacity or CPU resources are exhausted [104–106]. When retransmits are scheduled, it must be clear that the retransmitted packet can still arrive at the receiver on time and that the retransmit does not violate deadlines of subsequent, new and more important packets. This leads to an optimisation problem, which is often referred to as rate-distortion optimisation [1,107], because a packet schedule is optimised to achieve a minimal expected signal distortion at the receiver under a given rate constraint and an expected loss and delay probability of the channel. A packet scheduler that consider time, rate and signal distortion in order to selectively choose the best candidates effectively combines optimal adaptive error-control with rate-control.

2.3.4 Adaptive Rate and Congestion Control

Adaptive rate-control is required to cope with the time-varying bandwidth availability in best-effort packet networks. When the data rate emitted by a sender exceeds the available bandwidth of a network path, congestion and loss become likely. Hence, it is desirable to control the rate of a data stream. Fairness and stability of the Internet relies on rate-control performed at the end-hosts and hence, it becomes the task of the sender. Rate-control can be performed at the encoder level or at the transport protocol level, depending on the application setting.

Encoder Rate-Control Encoders can either operate in constant or variable bitrate mode. In the latter case, an encoder-internal rate-control mechanism ensures either a fixed rate at any point in time (e.g. for every frame) regardless of the source complexity, or an average but variable rate. VBR streams may temporarily exceed the average rate when source complexity is high or when intra-modes are selected.

Figure 2.3 shows a typical model of a video encoder. While the source signal consists of equally sized frames at a constant frame-rate and bitrate, the encoder may generate a variable bitrate and even a variable amount of data units per input frame. The bitrate can be controlled by an encoder-internal rate-control mechanism. Live-encoders can directly react to variable channel conditions, while offline-encoders must rely on an approximation of the channel bandwidth.

Network Congestion-Control Transport protocols can either in addition or alternatively to the rate-control of an encoder shape the sending rate by adaptively delaying data or by dropping data at buffer or deadline limits. This is to ensure fair bandwidth sharing with other flows in the network and to avoid congestion. The dominant congestion-control method in the Internet is that of the Transmission Control Protocol (TCP) [108]. TCP uses an additive-increase/multiplicative-decrease (AIMD) scheme which is not appropriate for media streams due to the rapid decrease in sending rate when congestion is detected. Hence, several approaches exist which try to be TCP-friendly on the one hand and

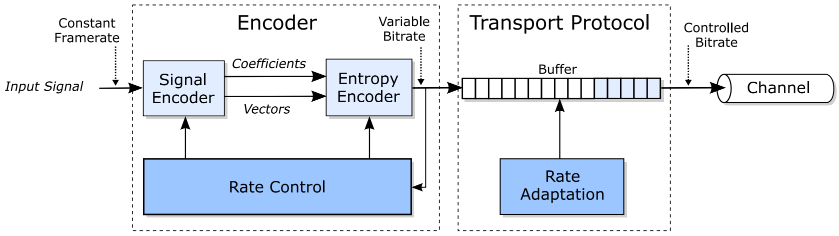


Fig. 2.3 : *Encoder rate control model [73].*

media-friendly at the other [109, 110, 110–112]. Prominent examples are equation-based schemes such as TFRC [59, 113] that mimic TCP-like behaviour, but provide a smooth sending rate.

Receiver-Driven Rate Control In receiver-based rate control, also known as layered quality adaptation or receiver-driven layered multicast [55, 114], clients control their receiving rate by adaptively requesting the server to add or drop layers from of a video stream or by adaptively joining or leaving multicast groups where layers are broadcasted by the server. Hence receivers can adjust stream delivery to their own capabilities and to network congestion they infer from loss in the layers they receive. Because layered quality adaptation is a coarse-grained adaptation strategy (the number of layers is limited for efficiency reasons) and because it requires a layered-encoded scalable bitstream it is not widely used yet. Multicast rate-control is still an active research area.

2.3.5 Buffer Management and Synchronisation

Receiver buffers are used to compensate for delay jitter introduced by the varying delay of best-effort networks, transport protocol scheduling and retransmission schemes. In order to reconstruct the temporal relationship between consecutive data units in a media stream, the output rate of receiver buffers must be controlled. This task is performed by playout schedulers (see [115] for a comprehensive survey) which either operate at a fixed rate or adapt its rate to avoid buffer over- and underflows. Synchronisation schemes can serve two purposes, *intra-stream* synchronisation to reconstruct temporal relations between data units of a single stream, and *inter-stream* synchronisation to coordinate the playout of different streams or flows of a common stream [18, 116].

Several buffer management schemes and playout schedulers have been proposed for different environments and application constraints, such as schemes that differentiate between talkspurts and silence in voice-over-IP applications [117], schemes for wireless cellular networks [118], and schemes that adaptively account for retransmissions while trying to minimise the required end-to-end delay [119].

2.4 Conclusion

From the current state-of-the-art in packet-based multimedia streaming we can draw the following conclusions. First, because the Internet seems to be the network of choice for multimedia communications, applications must deal with unpredictable and variable bandwidth, delay, and loss. Second, to achieve optimal quality in unstable network environments, streaming protocols increasingly rely on format-specific details of media streams. Moreover, optimal encoder performance requires knowledge about the channel characteristics. Third, the number and complexity of different streaming protocols is likely to overwhelm typical application developers. Programmers need simple abstractions and application-centric semantics to cope with the complexity of multimedia streaming. The current landscape of streaming protocols is too complex for the typical programmer.

While protocol standards such as RTP were not designed to provide appropriate control mechanisms, the existing error- and rate-control mechanisms and optimisation frameworks are often fixed to a single encoding format. Coordination between between media coders and transport layers is performed in an ad-hoc fashion that bundles encoders and protocols closely. This effectively circumvents reuse.

In response to these observations, we believe that a generic and content-aware middleware layer is an appropriate solution. A middleware can provide suitable programming abstractions and means to exchange necessary information such as channel conditions from the transport layer to the application and content-specific properties of data streams from the application to transport protocols.

Chapter Three

Related Work

To steal ideas from one person is plagiarism;
to steal from many is research.

(Steven Wright)

The work presented in this thesis bridges the gap between approaches from different research communities, the multimedia and systems community and the signal processing community. Hence, work related to this thesis is found in the literature of both areas.

In this section we briefly discuss recent developments in stream-based programming models and middleware platforms for distributed multimedia systems. We also focus on recent developments of content-aware media streaming protocols, importance estimation frameworks for media streams and cross-layer design approaches for network protocol stacks.

3.1 Stream-based Programming Abstractions

Stream-based programming models and frameworks to support the rapid development of stream processing graphs are popular in many information processing areas, such as packet routing and filtering [120,121], data mining [122,123], embedded systems design [124], and multimedia communications [17,125–127].

Because all frameworks intend to decrease the complexity of creating and managing stream processing graphs, they share the concept of composable modules, whereas composition is often expressed in a configuration language. Because components usually possess common stream interfaces, they can be easily composed into complex processing chains. Some connection semantics ensure data flow and call compatibility.

All component models for distributed multimedia systems provide three kinds of abstractions: one for processing stages, another for connections between processing stages and a third for transport units that flow through connections. Connections are often called bindings because they effectively bind two stages. A binding usually hides protocols and aspects of distribution from processing stages. Sometimes, an explicit application-level end-point, either owned by a binding or by a processing stage, is defined. Such end-points, often called ports, are used to identify streams and to provide buffering and synchronisation. A single port forwards a single media stream only.

While these frameworks are primarily concerned with graph composition and data flow management, they either lack remote interactions or encapsulate them into special components, hiding the details of communication. This is in contrast to our middleware that exposes communication, control, and cross-layer coordination to applications.

Click [120] targets the domain of network packet processing inside operating system kernels where it supports synchronous interactions based on method calls. While elements can support push and pull interfaces, Click ensures proper scheduling of activities and automatically checks for errors that inhibit correct packet forwarding in a system configuration. Multicasting is implemented by special components. The Click architecture couples components closely by method calls and lacks remote interaction because this is unnecessary in the intended application domain.

Infopipes [17] extend the functional composition aspects by abstractions to explicitly control concurrency and by components that perform remote interactions while completely hiding the complexity of network protocols. Infopipes are a generic high level abstraction for concurrency control in staged media streaming systems. Execution control is adaptively performed, based on dynamic observation and feedback. Infopipes encapsulate network communication in special components.

StreamIt [125] is a framework that consists of a stream programming language that allows a programmer to statically describe and connect complex stream processing graphs and a runtime-environment where adaptations of processing graphs are supported. Processing stages can exchange streams as sequences of timed unicast and broadcast messages through special inter-connection filters that may perform splitting and joining.

Xtream [126] is a high-level framework that provides novel abstractions of communication channels for distributed processing of multiple, not just continuous, media streams. Xtream channels establish loose bindings between connected components, support synchronised delivery and methods for automated selection of appropriate channel implementations based on desired semantics. Channels provide, however, no means to perform monitoring or cross-layer coordination with protocols.

Although our middleware shares some abstractions with these frameworks, we exclusively focus on content-aware delivery of data streams across networks. While component-based frameworks strictly hide all aspects of remote communication from their users to decrease application complexity, we believe that network-adaptive applications and middleware that exposes dynamic characteristics of channels are the appropriate way to build multimedia systems for unreliable packet networks.

3.2 Open Middleware Platforms and QoS-aware Middleware

The lack of QoS and multimedia support in object-based middleware platforms inspired many research projects to explore ways for integrating these aspects into multimedia systems [128], multimedia component frameworks [129] and open middleware architectures [10–12,37,130–133]. These platforms mainly focus on quality-of-service aspects, mobility and adaptation issues. The main objectives are the hiding of complexity imposed by distributed environments, QoS mechanisms and adaptation procedures.

An early result of the research on open middleware was the definition of the **Reference Model for Open Distributed Processing (RM-ODP)** [134] as ISO standard. RM-ODP provides a common and abstract framework for distributed system platforms. It defines operational interfaces for RPC and RMI interaction styles, signalling interfaces for asynchronous event-based interaction styles, and typed stream interfaces. The interaction model of stream interfaces is a flow which is represented by a sequences of interactions. Unlike operational interfaces, stream interfaces require explicit bindings. Bindings are encapsulated by binding objects which allow for the control and inspection of the binding and its QoS parameters.

A fundamental design decision of open middleware platforms is that bindings can expose some details of their implementation. Reflection concepts are used to infer the signatures of exported control interfaces and adapt the middleware implementation to application requirements. **TOAST** [10], for example, incorporates reflection concepts into binding objects to support adaptation on an architectural level. Reflection allows for inspection of objects, but also for fine-grained configuration changes and even structural changes. Changes can range from altering interfaces and the behaviour of objects or components to the complete replacement of components. Although reflection mechanisms are powerful tools to investigate and monitor single objects, applications are required to know the semantics of the inspected and altered object implementations. Hence, it remains a challenge to control and adapt advanced streaming protocols, in particular the complex error-control and flow-control mechanisms, by means of reflection alone.

Other middleware architectures specifically address quality-of-service support in distributed computing that is not only targeted at multimedia applications alone. **The ACE ORB (TAO)** [16], for example, explores architectures and interfaces for hard-realtime and low-latency distributed computing in general. TAO's architecture is mostly oriented on the CORBA programming model and remote procedure calls. A multimedia extension to enhance TAO [19] by media streaming following standard recommendations does only deal with multimedia device control via RPC rather than data transport via streaming protocols. **Quality Objects (QuO)** [14, 15] also aims at supporting QoS for RPC-style distributed systems, but specifically develops techniques to adapt QoS for bindings between remote objects at run-time.

There are similarities, but also important differences between this work and ours. A multimedia streaming middleware can clearly benefit from research results in real-time middleware support even if the basic interaction patterns differ. There is, however, a fundamental difference between RPCs and media streams that requires us to reconsider basic design decisions for QoS and adaptation support. Media streams are not always required to arrive at a sender without loss. Certain parts of a stream may be lost in transit or dropped at the sender without considerably degrading application performance. RPCs, in contrast, require full reliability which renders them much less appropriate for run-time optimisations. Overall system efficiency may benefit from the additional freedom media streams introduce. This comes, however, at the cost of extra optimisation models that need to select whether and when to transmit data units in a stream.

3.3 Multimedia Middleware Platforms

While open middleware platforms support multiple personalities and interaction semantics, the platforms discussed next are exclusively designed for multimedia applications and multimedia data stream delivery. The concepts of bindings and ports are often used by these platforms, but most of them focus on special aspects only. **DaCaPo++** [13], for example, explores architecture support for flexible protocol composition and security, **Cinema** [34] is concerned with intra-stream and inter-stream synchronisation, **MUM** [43] concentrates on handover schemes in mobile wireless networks. The difference to our work is, that these middleware systems assume unstructured real-time media streams that contain data units which are equally important. Hence, there is only need to express packet deadlines rather than exploiting other content-specific attributes during a streaming session. The mentioned systems cannot support network-adaptive applications because they do not provide feedback about channel conditions.

DaCaPo++ [13] is a multimedia middleware platform that allows for the composition of streaming protocols from small building blocks in order to support a variety of protocols and networking technologies. DaCaPo contains an efficient protocol execution engine, provides QoS-aware resource management and security features. Protocols are organised as a protocol stacks and applications access connection end-points of the topmost protocol for data delivery. Applications may configure protocols and build protocols stacks at session-setup time. Inspecting protocols at run-time and monitoring transport channels is not supported. Although the decomposition of protocols into small building blocks is promising because it suggests that protocol functions are reusable, the actual composition of complex streaming protocols quickly becomes infeasible due to their complex interactions.

The **Network-integrated Multimedia Middleware NMM** [20] is an efficient and extensible platform for distributed media streaming and control of multimedia applications. NMM uses explicit end-points and bindings to connect stream processing stages, supports stream format negotiation and automated component instantiation. NMM defines a combined transport and signalling protocol to deliver data units and signalling messages. The control of remote components as well as remote connection setup between two stream end-points is performed via remote procedure calls. The interaction semantics and design of NMM are very similar to our middleware approach, which shows that our communication concepts seem to be a widely accepted solution among users. In addition to the basic concepts of ports and bindings, our platform gives programmers detailed control over binding semantics by decoupling aspects of error-control from interaction and topology management. In addition, our platform focuses on application-protocol coordination which is not available in any of the above platforms.

Streaming server platforms such as **NetMedia** [135], **Yima** [136], and the **Darwin Streaming Server**[1] provide solutions for the scalable delivery of pre-encoded data streams to multiple receivers. Besides scalability, data placement on disks, and failover handling these systems include protocol support for flow-control, congestion-control and partially reliable delivery. NetMedia [135], for example, is a middleware for on-demand scenarios and pre-encoded streams that runs on streaming servers and streaming clients. NetMedia integrates network congestion control with end-point buffer management and stream synchronisation. Adaptation is performed in reaction to feedback mechanisms. While the application scenario of these systems is on-demand delivery and a single or a limited set of streaming protocols is available only, our platform is capable of supporting arbitrary scenarios, delivery topologies, and protocols with a single abstract programming model.

Several other research middleware platforms target application-level QoS management support as their primary objective. They are intended to assist programmers and maintainers of large-scale distributed multimedia installations to specify and control the QoS of components. **QCompiler** [137], for example, is a high-level specification and translation framework that uses meta-data descriptions to express QoS requirements. A meta-data compiler generates translation modules that can semantically negotiate QoS between components even if components have diverse quality semantics. **Spidernet** [138] and **Hourglass** [139] aim at the distributed composition of complex application functions based on user preferences and resource availability. The basic difference between these works and the work presented in this thesis are the level of abstraction and the time-scale at which decisions are made. While QoS management focuses on application-wide service composition and operates at session scale, our work focuses on low-level protocol coordination that considers single data packets or groups of packets.

[1]`http://developer.apple.com/opensource/server/streaming/index.html`

3.4 Closely related Multimedia Middleware

There are only a few directly related middleware approaches: Priority Progress Streaming [104], Horde [140] and M-Pipe [141].

Priority Progress Streaming (PPS) [104] targets video-on-demand streaming over the Internet. It combines a scalable video encoding format (SMPEG) with a priority mapping algorithm that is aware of SMPEG bitstream features. A content-aware scheduler uses the priority information to drop unimportant data units when bandwidth availability decreases. The importance of a data unit is calculated based on (a) the type of the data unit (e.g. I, P, or B-frame), (b) the encoding layer of the data unit (only spatial scalability is supported), and (c) a user provided adaptation policy. The policy can either favour drop in the temporal dimension which results in a lower frame-rate or the spatial dimension which results in a lower resolution. Calculation is performed over all data units in a scheduling window and usually resembles the partial dependency order of units. The order is determined by a drop-before relationship according to the dependencies generated by the scalable video coder. The content-aware scheduler finally combines the priority value with a deadline value to decide about dropping a particular data unit at a given rate constraint.

This scheme very closely resembles the ideas behind rate-distortion optimised packet scheduling (see section 3.5), except the fact that dependencies between data units are used only. In this sense, PPS is similar to our approach of calculating data unit importance based on dependency relations. PPS differs, however, in the following points. First, PPS assumes a fixed encoding format with a-priori known types and semantics. Second, the structural information about dependency between data units is only used during priority mapping. In contrast, the content-awareness framework we propose in chapter 4 is independent of encoding formats and allows to analyse dependencies in more ways than just to derive importance. We explicitly extract dependency information and make the dependency graph available to arbitrary algorithms.

M-Pipe [141] defines a framework for generic format-independent media adaptation of scalable media streams that is explicitly designed for network layers or intermediate network nodes. M-Pipe uses cross-layer coordination to signal properties of data units by explicitly labelling data units with a layer-independent descriptor (LID) [142]. At connection setup, a descriptor can be exchanged with lower layers to define operation points for later adaptation. The descriptor can contain a desired traffic class, packet drop preferences and error protection preferences. This enables protocols to perform unequal error protection and expedited forwarding of data units. The general idea of attaching labels to data units is similar to the hinting concept in our content-awareness framework. Our approach differs in the set of information that is contained in labels. While M-Pipe defines attributes that express information for a particular data unit only, we include information that can express relations between data units. Based on this information our framework can detect if dependencies are satisfied and even if a required data unit is missing (we introduce this concept as *broken dependency* later).

Horde [140] is a middleware architecture that supports flexible striping of flows over multiple network interfaces with cross-layer coordination. Horde applications can control striping and packet scheduling at the transport layer by means of QoS objectives. QoS objectives express the utility of data units for the application and a constraint on loss probability and latency. Horde supports separate objectives for data units of different types, such as I, P and B-frames in video streams. To increase usability, the objectives are expressed in a description language at design time of an application. At runtime, a packet scheduler considers the current channel state (loss ratio and delay) as well as the application-specific utility of data units in its decision. Packets are only sent over a particular channel when the channel meets the desired objectives. Because finding optimal solutions based on such objectives becomes computationally impractical, the utility-based packet scheduler uses windows (transmission slots) to lower the complexity. Again, the concept of using relations between data units is similar to our approach. However, Horde restricts itself to MPEG-like stream structures of I, P, and B-frames, while our content-awareness model is able to express arbitrary relations and even groups of data units that belong to the same frame, layer or description.

3.5 Content-Awareness in Streaming Protocols

Different properties of media streams are used to perform content-aware tasks in advanced transport protocols. Examples are the importance of data units, the dependency pattern between data units and the signal distortion generated when a data unit is lost. Importance metrics can be used to select whether and when to transmit and repair data units and which amount of error protection to add. Rather than streaming the packetised media in a fixed sequence according to presentation time, a sender can choose a different transmission policy that favours important data units while maintaining a transmission rate constraint. More important data units can even be pre-transmitted far in advance of their presentation time, while less important data units are transmitted later or not at all. In the following we give a brief overview of recent approaches to increase robust delivery of media streams over best-effort packet networks:

Content-aware protocols require a notion of data importance to efficiently protect data units from transmission errors. Based on this information, protocol mechanisms have been proposed that perform selective drop [104, 105], packet interleaving [102, 103], unequal error protection and forward error correction [78, 84, 86], packet scheduling and selective retransmissions [1, 89, 143] as well as priority mappings to Quality-of-Service classes [144, 145]. These protocol schemes have shown that the overall signal quality at the receiver significantly improves when information about expected channel errors and the contribution of a data unit to the reconstructed signal's quality is available at transmission time.

Some approaches in the literature deduce the importance of data units directly from their types [145, 146] or the position of a frame in the group of pictures [100, 104]. While such static classification schemes are state-less and easy to implement, they lack a loss history, cannot predict the importance of lost data units and the conditional importance in fragmented and multiple description streams. Our content-awareness framework supports these features.

Isović proposed QAFS [147], a **Quality-aware Frame Selection Algorithm**, used for skipping frames during decoding of MPEG-2 streams in order to achieve guaranteed decoder runtime on resource-constrained systems. QAFS estimates the importance of frames according to multiple properties, such as the frame type, the frame position in the GOP, the frame spacing, size and buffering constraints. The lowest importance is assigned to (small) B-frames because they are not used as references in MPEG-2 and they contain less information. While QAFS is targeted at MPEG-2, the assumptions of fixed GOP structures and known dependency patterns do not longer hold for advanced encoding schemes such as H.264, MDC, FGS and SVC.

The most prominent technique is the **Rate-Distortion Optimisation model (RaDiO)** [1], that formulates the packet selection problem as a multi-parameter optimisation problem. Various R-D models have been proposed in the literature [2,93,105,148–153]. They usually assume knowledge about dependency relations in the stream and often also the error-concealment strategy used at the decoder. R-D models rely on distortion metrics which are defined in the signal domain. Distortion of a data unit expresses how much noise is removed from a reconstructed signal when the unit itself is available for decoding. Although distortion metrics are known to produce accurate importance estimations, obtaining distortion values is expensive. That is because distortion is usually calculated as the mean squared error (MSE) between an error-free and an erroneous reconstruction of a signal. These calculations are expensive because they require multiple decoding simulations with different loss patterns for every single data unit. In real networks individual loss affects network packets, but it depends on the packetisation scheme which application-level data units are actually lost. Authors usually abstract from packets by considering loss at the frame-level [89,105,143,148] or the macroblock level [144]. Still, the computational complexity of RD models is high. In order to limit the complexity several approximation models have been proposed [148–150], which are known to yield acceptable results. Inter-frame dependency for RD-models is modelled as a directed acyclic graph (DAG) [154, 155] or as partially ordered set [102, 104], whereas both notations are convertible. Röder et. al. [155] show that rate-distortion optimisation is NP-hard and extend the work of [1,150] by branch-and-bound that are more efficient and that surprisingly generate more precise packet schedules.

Although distortion models are known to yield accurate importance estimations, some problems remain. First, each R-D model is specialised for a single encoding format and a specific concealment scheme. In several scenarios information about employed concealment at a receiver is either unknown or differs between receivers (e.g. broadcasting and multicasting setups). Second, dependency structures of modern encoding schemes are no longer fixed. Instead, an encoder is free to choose the most suitable reference pictures and when feedback is available, the encoder may even adapt the reference picture selection to the observed loss pattern. Third, the abstraction from packetisation rules is inappropriate especially for streams at high spatial resolutions that tend to have data unit sizes which are even larger than a datagram payload size allows. Because these problems limit the utility of R-D models in such environments, an extension is required. Our generic dependency framework presented in chapter 4 can serve as such an extension because it considers fragmentation and dependency relations of arbitrary encoding formats. It can be easily integrated into R-D models because they model dependency as a discrete operator without assuming implementation details.

Chapter Four

A Framework for Content-Aware Media Streaming

Prediction is very difficult, especially
if it's about the future.

(Niels Bohr)

In this chapter we develop and analyse a content-awareness framework that enables system layers to access importance distribution and structural properties of data streams. Although the framework is principally designed for continuous multimedia streams, it is not restricted to this application class or a particular encoding format.

We start this chapter with discussing the utility of content-awareness in system layers and present general observations about properties of media streams that may be exploited for content-awareness. Based on these observations we construct a dependency model that can validate the structural integrity of streams and estimate the importance of data units in streams. We propose a description language to express static properties of dependency patterns and present a prototype implementation of a validation and estimation service that can be embedded into applications and system layers.

Extensive simulations with real video streams encoded with different dependency structures reveal that dependency-based importance compares well to traditional distortion metrics. Performance results indicate that the prototype of our model can estimate importance values of several complex and up to thousands of simple streams concurrently in real-time in a realistic packet-scheduler scenario.

4.1 Content-Aware System Layers

We are basically concerned with improving efficiency and robustness of streaming in layered communication architectures. Our main idea is to share certain information about content-specific properties of requests and data units between applications and network protocol layers because this would allow protocols to perform more educated error-control and scheduling decisions. Surely, not every protocol layer would understand application information or can care about it for scalability or layering reasons. Hence, we are looking for a generic way of passing information without changing layering assumptions of protocols.

Therefore, we choose to extend protocol stacks by facilities to pass meta-data as *hints* to lower layers [156, 157]. Hints are a conceptual extension to normal data passing calls which add a pre-defined set of meta-data to every payload data unit. The meta-data is forwarded between protocol layers along with the payload when a call is processed. Hints are optional, that is if a hint is used at some layer to perform scheduling or error-control operations, the application may benefit from extra robustness and efficiency. If a hint is not used, the application gets the same service as if no hint was given at all. Any protocol layer is free to access and copy hints. Unaware system layers can safely ignore and even drop hints without breaking application assumptions. This semantics is in accordance with the end-to-end argument [158], which states that applications may make no assumption about whether, where and how a function is actually implemented at lower system layers. Thus, hints do not increase system complexity or jeopardise robustness and security which makes them an efficient and safe alternative to dynamic extension mechanisms [159, 160].

Although this thesis uses hinting to enhance network processing, hints are not only applicable to protocol stacks. They may be used by any system layer that directly or indirectly processes application data and requests, such as filesystems and I/O devices. As an example consider current filesystems. They treat data units in media streams equally. At most they employ per-process or per-file heuristics to adapt caching and prefetching behaviour. Content-awareness would enable a storage stack to use per data unit hints to improve disk layout and adapt request scheduling policies. A content-aware disk scheduler can, for example, skip reading unimportant data units when a stream is fast forwarded or replicate important data units for faster read access.

In the following we assume that data units in a media stream are unequally important, because some contain essential and others optional data. Unaware network protocols would treat all data units equally, basing scheduling decisions on system wide fairness or throughput objectives. A content-aware protocol layer would in addition to these objectives use the different importance values attached to data units to prioritise essential data and probably drop optional, less important data. Such adaptive services are in particular useful when applications are partially tolerant to the loss of some data such as video streaming applications. Consequently, system-level schedulers gain more degrees of freedom for their scheduling decisions and applications may experience increased efficiency, improved response times and improved error resilience.

A special class of adaptive and partially loss-tolerant applications are digital multimedia streams, in particular because the importance in such streams is unequally distributed. Importance in media streams is a dynamic and multidimensional property. It depends on the amount of signal information contained in data units rather than the number of bits. Importance is also influenced by the dependency between data units, the loss history experienced by a stream receiver and the deadlines of data units. For example, units that contain essential information that is referenced by many other units are more important, while unreferenced or late units are less important. A content-aware communication stack can exploit these properties to perform one or more of the following tasks more efficiently:

Adaptive Packet Scheduling: Content-awareness enables a scheduling policy to access which impact a particular drop or scheduling decision has on subsequent data units and on the application-perceived quality. Packet schedulers that select data units for transmission and retransmission in priority order rather than encoding or display order can increase the probability of successfully delivered important data units in favour of unimportant units. Similar effects are achieved by selective drop policies [104] which systematically discard data units to handle missed deadlines, buffer overflows and network congestion.

Selective Error Control: Content-awareness enables transport protocols to become selective and more efficient when performing retransmissions, packet interleaving, and forward error correction [3, 22, 23, 83]. Error protection can systematically add more redundancy to more important data units while content-aware packet interleaving can reorder adjacent packets of high importance to reduce damage imposed by burst-loss. Retransmissions can become selective, i.e. a protocol can decide whether, when and how to retransmit a data unit based on dependency, importance, and deadlines.

Priority Mapping: Quality-of-Service based networks already allow applications to specify an appropriate service class for their traffic to enable expedited transport. Content-awareness can be used to classify data units individually, rather than per connection. An application can associate dependency information, importance values and deadlines at the packet level while priority schedulers at lower protocol layers are able to assign resources more efficiently and adapt scheduling decisions to the actual packet content (e.g. [103]).

Content-Aware Scaling: Wireless networks, multicast environments and mobile devices impose resource and connectivity restrictions on streaming applications. Different receivers may have different display resolutions and are likely to observe variable network conditions. Scalable encoding schemes already enable applications to deliver a single bitstream to a diverse set of receivers, while the actual adaptation of the stream is performed at active network elements, such as wireless base-stations and proxy servers. Content-awareness allows to design such in-network scaling services independent of particular data formats (e.g. [104]).

We do not intend applications to attach arbitrary meta-information as hints, because this would require additional coordination between layers to negotiate the semantics of hints. Instead, we identified general parameter sets for the domain of real-time media streaming. Before we are going to discuss these parameter sets, we first define general requirements a parameter must fulfil to be useful in hints.

4.2 Objectives and Challenges

A variety of information about data units in media streams can be used for content-awareness. There are, for example, structural information about data unit types and dependency relations, timing information such as deadline, duration and synchronisation points, as well as error-resilience information about important and unimportant bits in a data unit. As a first step towards content-awareness, we develop a dependency model which focuses on structural properties, in particular, the dependency relations between data units. Our model will provide a generic mechanism to track dependency between data units and infer their dependency-based importance. Since time and error resilience issues are orthogonal to dependency, we will integrate them later.

The main objectives of the dependency model are structural validation and importance estimation at a very low computational costs. Our main requirements are:

Format Independence: In order to serve as a universal foundation, the dependency model must be able to track the complex dependency patterns found in media streams regardless of the stream format and the packetisation level (e.g. video objects, frames, slices, or network packets).

Purpose Independence: The dependency model must be generally applicable at different system layers of stream senders, receivers and proxies. The model must not make assumptions about available information which cannot be met by a particular layer. In addition, the model should not make assumptions about the usage pattern, in particular, about when and in which order data units become visible and when and how often estimations are required because this may differ between the various modules of streaming protocols such as scheduling, error protection, fragmentation, and scaling.

Efficiency and Predictability: Since media streaming systems operate close to the resource limits the additional costs for dependency tracking and reasoning must be justified by the gains. Costs arise from the amount of initialisation data which is exchanged prior to a stream, the amount of extra meta-data attached to data units, and the computational overhead of dependency tracking and importance estimation. Since many media streaming applications are delay sensitive, the dependency model must not introduce unpredictable delays to protocol processing and stream forwarding.

Robustness: The framework must be robust against loss of meta-data due to uncontrolled packet loss (isolated and burst loss), controlled packet drops and packet reordering. The framework may give weaker information due to incomplete knowledge, but must recover quickly as more data becomes accessible.

Instead of inferring the importance of data units from their contribution to signal reconstruction quality as done by traditional distortion metrics, our model estimates importance based on dependency between data units alone. This significantly decreases complexity, but may also yield less accuracy compared to existing solutions. In the evaluation we will show, which effects actually contribute to a decrease in accuracy.

Dependency tracking is challenging, since the network may lose and reorder packets and future references may be invisible. One might wonder why we even aim to model dependency when there is another obvious and simple solution for expressing importance. Similar to selecting a traffic class in a multi-QoS network, one might attach a static priority value to each data unit and select a corresponding policy at the transport layer for unequal error protection and packet scheduling. Simple types of this feature are supported in H.264/AVC as ref_idc values and in JPEG2000 as discardable flags. While this approach seems simple and sufficient, it ignores that importance in media streams is relative between data units and varies with the context. Data unit importance dynamically depends on deadline and size, the availability of other data units, and the history of already transmitted or lost units. Data units become, for example, useless if they miss their deadlines or if a unit containing referenced data is lost. A static solution, where priority is deduced once from data unit types yields results which are far more imprecise. In contrast to static solutions, importance estimation requires a dynamic solution that tracks at least dependency and transmission history and optionally even content-based distortion values. This chapter presents such a solution and shows that it is efficiently implementable.

4.3 Structural Properties of Media Streams

The aim of our content-awareness framework is to support the reasoning about dependency relations and importance distribution in media streams. In order to sufficiently understand the problem domain and available design options, we will first concentrate on general properties of different streaming formats.

4.3.1 Quality and Distortion

A common technique to obtain the actual importance of data units in media streams is to measured or estimate the *expected distortion* when a particular data unit is lost. The distortion defines the difference between a reconstructed signal and an error-free reference signal. Often, the mean squared error (MSE) metrics is used to quantify the distortion. Because from the point-of-view of the transmitter the real distortion at the receiver is unknown (it depends on the actual packet loss pattern in the channel and the employed error concealment strategy at the receiver) a transmitter can only approximate the distortion, while a receiver sees only one of many possible reconstructed sequences, depending on which packets are actually lost. Therefore, the actual distortion at the receiver does not necessarily equal the expected distortion, but averaged over all possible loss realisations, expectations become statistically reasonable.

Several methods for calculating the expected distortion do exist. They either accurately compute the per-pixel distortion or use models to estimate it [1,148,149,161]. Although distortion metrics are known to yield perfect importance estimations, obtaining distortion values is expensive. It requires multiple channel simulations and is limited to pre-encoded content [162]. Hence, models are used to approximate the real distortion values [107]. Distortion models often assume special encoding and concealment schemes, fixed dependency structures, and a one-to-one mapping between encoded data units and transport units. This limits their utility to special stream classes and network environments. Although current distortion approximation models have low computational complexity, several assumptions made by the models do not easily translate to real system environments:

- Most distortion models assume a fixed concealment method, but the actual method employed at the receiver is typically unknown in real settings. In multicast and broadcast scenarios, the concealment method may even differ between individual receivers.
- In order to remain tractable, distortion models assume one-to-one mappings from application-level data units to network packets. Resilient delivery of media streams over packet networks requires, however, proper fragmentation of data units. Our experiments revealed that even at very small resolutions (QCIF), encoded video frames require fragmentation because they become larger than the network MTU size. In this respect, distortion models abstract too far from reality.
- Existing distortion models also abstract from actual dependency relations, but they provide no general means to express and track

dependency patterns. Usually the existence of a mapping function is used to determine predecessors of a data unit is assumed. Because advanced video coding schemes contain an increasing number of features that generate irregular dependency patterns (e.g. adaptive reference picture selection, adaptive intra-refresh and weighted prediction) a dependency tracking model is even required by traditional distortion models to implement this mapping.

These practical problems indicate, that although distortion is a perfect measure for obtaining the importance of data units, a practical content-awareness framework requires a low complexity alternative. A promising approach is the tracking of dependency relations in media streams because dependency is the main reason for error propagation in predictive coded sequences.

4.3.2 Dependency Relations

Multimedia streams are continuous and strictly ordered sequences of logical data units [163]. While timing and order of these data units is a property of the contained audio-visual information itself, dependency between data units is introduced by encoding techniques. Although our primary focus is on structures that were generated by predictive block-based video encoders, we regard dependency as a general concept that may equally apply to other encoding schemes, such as wavelet encoders.

Multimedia encoding standards define data units as instances of types, based on a format-specific type hierarchy. H.264/AVC [27], for example, defines seven types at the slice level (I, P, B, EI, EP, SI and SP-slices) while MPEG-4 Visual defines three types of arbitrary shaped video objects (I, P, B-VOPs). Each video frame (or picture) may consist of multiple slices or video objects, and multiple frames are again combined into picture groups[1]. These recursive patterns are continuously repeated within a sequence. Sometimes, dependency can only be expressed across several abstraction layers as in H.264/AVC. Even if the unit of processing and packaging is a slice in H.264/AVC, predictive encoding introduces dependencies between a slice and preceding reference pictures. When pictures are composed of multiple slices, each slice effectively depends on all slices that contain data of referenced pictures.

[1]The Group-of-Picture (GOP) concept was originally introduced by MPEG-1 as a structural composition element. Later standards superseded it, but because of its expressiveness it remained in use for illustration purposes.

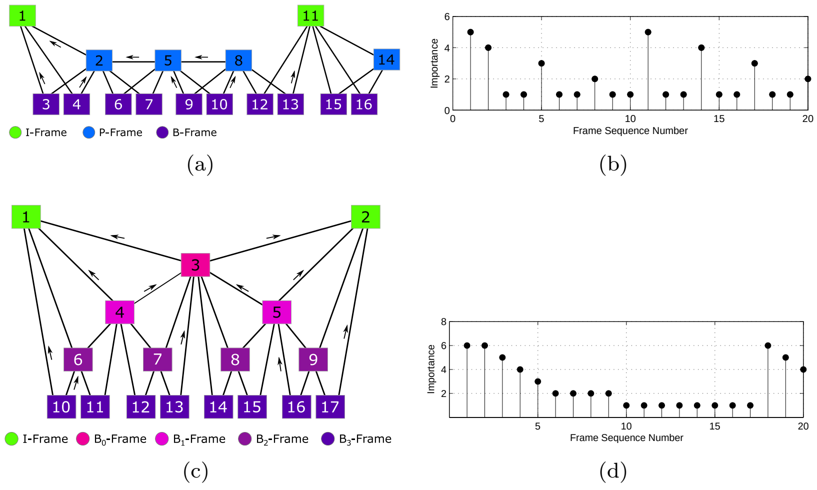


Fig. 4.1 : *Examples for 4.1(a) a traditional I-P-B structure and 4.1(c) a pyramid structure used for temporal scalability in H.264/SVC. Figures 4.1(b) and 4.1(d) display the dependency-based importance values per frame.*

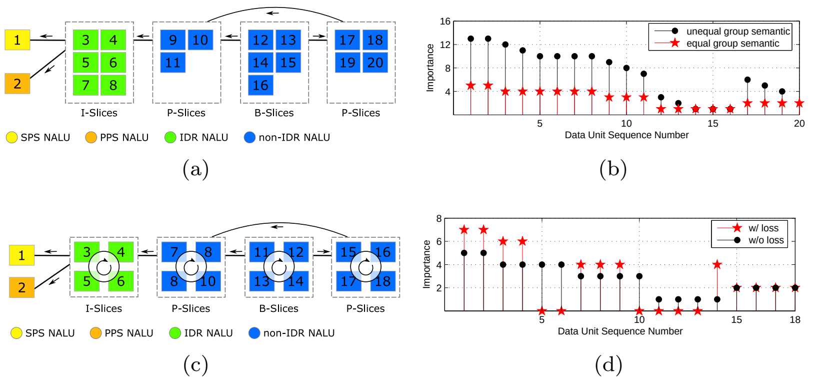


Fig. 4.2 : *Examples of fragmented H.264 streams: In 4.2(a) H.264/AVC NAL units contain equally important fragments (slices of one frame). In 4.2(c) mutual refinements of a single frame are encapsulated into multiple NAL units. When a unit is lost, the remaining group members become more important (e.g. when units 5, 6, and 10–13 are lost, the importance of units 1–4, 7–9, and 14 proportionally increase with the number of missing group members. Figures 4.2(b) and 4.2(d) display the dependency-based importance values.*

Efficient encoding techniques introduce increasingly complex dependency relations between data units. Figure 4.1 displays two popular examples of such structures. In MPEG video (fig. 4.1(a)) predicted frames can depend on intra-coded frames and other predicted frames, while bi-directional predicted frames are never used as references. Hence, B-frames can be lost or dropped without severe quality degradation, while lost I/P-frames have much larger impact on reconstruction quality distortion. Figure 4.1(c) shows a multi-level dependency structure for temporal scalability. Here, B-frames form a pyramid-like tree that generates evenly spaced gaps when dropping one or more hierarchy layers during scaling.

H.264/AVC (fig. 4.2(a)–4.2(d)) fragments stream data into slices, packs slices into *Network Adaptation Layer Units* (NAL units) and defines bi-directional, multi-picture, weighted dependency and long-term relations [164]. Here, groups of NAL units may have a similar type while they contain information of different importance and dependency. Moreover, all the NAL units of a single frame may depend on all NAL units of another frame. Partitions of different importance can be generated and transported in NAL units of different types. The H.264 joint-scalable video coding extension (SVC) goes even further and introduces refinement layers in multiple dimensions (spatial, signal-noise ratio, and temporal refinement layers).

An important observation about dependency in media streams is that bitstream layouts avoid backward-dependency loops to keep decoder complexity and memory management costs low. Even if future frames (in display order) are used as references during encoding, a coder reorders the data units so that referenced frames are sent prior to depending units. This concept is known as the stream's transport order. Hence, dependency relations always form a directed acyclic *dependency graph* [153], which is an important mathematical property exploited later in our dependency model because this allows us to easily follow dependency relations between data units.

Dependency is a central property of media streams that directly affects the importance of data units. The more units depend on a given data unit the more important it becomes for the reconstruction of the original signal. This is because if the referenced unit is lost, the resulting reconstruction error propagates to all depending units. Figures 4.1(b), 4.1(d) and 4.2(b), 4.2(d) display examples for importance value distributions. The first data units in a GOP are the most important ones because all subsequent units use them as references.

So far, we intuitively used dependency to describe that the existence of one data unit is *essential* for processing a related unit. This is sufficient when data units contain complete frames, but fragmentation, data partitioning and multiple description coding (MDC) [165] require different semantics. Consider the above mentioned H.264 example, where each slice depends on all slices of a referenced picture. All slices are equally important here, because every slice contributes to the frame's quality. MDC, in contrast, splits the encoded signal of each frame into groups of independently decodable units, called descriptions. The relation between those descriptions is a mutual refinement rather than a unidirectional existence requirement. In such schemes, every data unit of a description group contributes a small amount to the increase in fidelity of the reconstructed signal if present at decoding time. If absent, the remaining descriptions become more valuable because at least a single description is necessary to decode a signal at all. Figure 4.2 shows how importance dynamically changes if losses occur in such streams. In order to express dependency relations in fragmented streams we will later use the concept of groups to combine fragments and group semantics to specify a particular relationship inside a group.

Often, dependency is a fixed property of data unit types. Here it is sufficient to infer the actual dependency between data units *implicitly* from their type. In contrast, adaptive encoding schemes that generate dependencies based on content-specific attributes require *explicit* dependency relations per data unit. Type-based dependency is still a necessary requirement, but alone it is no longer sufficient. Figure 4.3 displays two

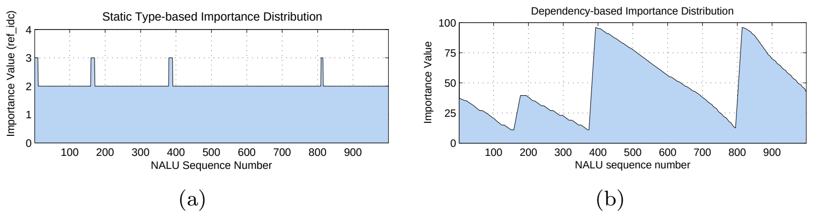


Fig. 4.3 : *Static and dynamic importance distribution in a typical H.264/AVC bitstream at HD resolution and with robust slice partitioning. 4.3(a) shows the static type-based priority (the H.264 ref_idc values) while figure 4.3(b) displays the dependency-based importance of the same stream.*

importance distributions for the same H.264/AVC stream, one based on static `ref_idc` information in NAL unit headers (left) and another that displays dependency-based importance between the same NAL units (right). Note that `ref_idc` values are expressed in two bits only and that a value of three denotes the highest importance. Dependency-based importance is, in contrast, more fine-grained, but similar to H.264/AVC it defines zero as the lowest value too. From this example it becomes obvious that static values can hardly express the real importance of data units. We could have alternatively chosen the NAL unit type to infer static importance values, but the picture would remain the same, because H.264/AVC defines only two different types to carry encoded picture data (see also section 4.7.2).

Dependency relations can span considerably large sections of a stream such as several GOPs or even the total sequence. Consider, for example, stable background images or sequence and picture parameter sets in H.264 [49]. While all frames in the complete sequence rely on their information, they are either sent once at the start of the sequence or they are repeated as in DVB. We call such dependency relations *long-term references*. Because long-term references can be referenced by any subsequent data unit, a decoder must store them until the sequence ends or they are explicitly updated or removed. Because long-term references increase storage requirements for decoders their number remains typically small. H.264 assumes a maximal number of long-term references which is defined at stream start-up time. Implementations often limit this number to 16 frames and a small number of parameter sets. In contrast, *short-term references* span only a limited distance within a stream, such as a single GOP. We call the maximum of this dependency distance the *dependency radius*. It will be defined later for every data unit type.

The dependency radius of types is usually limited to small sections of a complete sequence. There is, for example, no dependency across distant GOPs and only limited dependency between frames in adjacent GOPs in MPEG-1/2/4 (the last B-frames of an open GOP in MPEG video depend on the first I-frame of the following GOP). Even within a GOP there is limited dependency between frames. Consider, for example, MPEG-2 B-frames which may depend on the last and the next P-frame, but not all P-frames in a GOP. These restrictions will later become useful for constraining relations in our model.

An important observation in real streaming systems is that the dependency radius usually encompasses data units on distinct processing nodes in a network. Especially when very low delays are required or buffers are limited, one data unit may already have been decoded and displayed at a receiver node, while dependent units are still generated by a sender node. Consequently, it is desirable to decouple actual stream processing from reasoning about dependency. Therefore, we clearly distinguish between data units and meta-data about them. Meta-data can be stored and processed separately without delaying data units. This enables us to keep historical information about the stream without requiring the actual payload to be available anymore.

4.3.3 Visibility and Predictability of Structure

In real systems the visibility of data units is limited, either because a stream is generated live or data units are dropped, reordered or lost, either intentionally or by random packet loss. A sender is only aware of data units that have been already generated while a receiver can only have information about successfully received units. Type and dependency of lost and future units is unknown, we call it *invisible*. Figure 4.4 depicts the concept of visibility from the viewpoint of an arbitrary stream processing stage such as a streaming server, proxy or client application. We call the section of the stream between the earliest and the latest visible data unit the *horizon* [166].

Ideally, the horizon is continuous, but in practise it contains gaps due to drop, loss or reordering of data units. As time advances, new data units become visible to the processing stage while old data units are removed from the horizon due to resource restrictions.

When a data unit is lost, the missing information causes signal reconstruction errors, that may propagate to subsequently predicted data units. In order to track error propagation effects in dependency chains we introduce the concept of *broken dependency*. Broken dependency reflects the situation that at least one transitive predecessor of a data unit in the dependency chain is missing, but it does not unveil the reason why it is missing.

Invisibility creates uncertainty about the real properties of data units, including their type and their actual dependencies. For this reason lost data units and unknown future units may influence the accuracy of our dependency model. We will later discuss relevant effects in section 4.9. Hence, it is desirable to predict importance or dependency. While impossible in general, prediction is feasible under certain constraints discussed in detail in section 4.6.2. Here, we use three main classes of stream formats that differ in their predictability properties [166]:

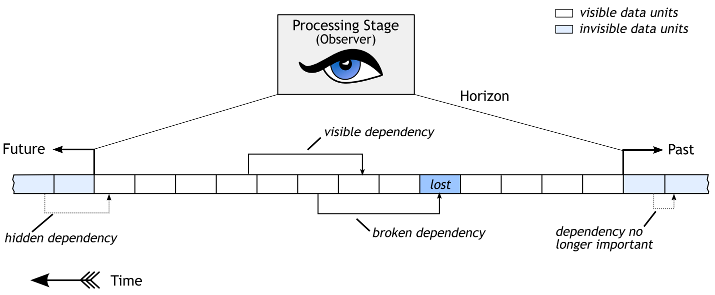


Fig. 4.4 : *Visibility of data units and dependency relations.*

Strictly predictable streams contain only data units for which type and dependency is predictable from sequence numbers, regardless of the horizon. While such streams are restricted to a fixed structure, they benefit from perfect importance estimation that is independent of visibility. Even dependency and importance of lost data units is perfectly reconstructable. Examples of strictly predictable stream formats are Digital Video (DV) and certain MPEG profiles with fixed GOP structures (e.g. MPEG-2 for Digital Video Broadcast - DVB, or MPEG-2 studio profiles that utilise I-frames only).

Limitedly predictable streams possess dependency patterns that may be variable, but known before data units become visible. This can be accomplished by embedding dependency information into the stream prior to the involved data units or by specifying predictable properties for a subset of types (e.g. every 12th frame is intra-coded). While options for limited predictability are manifold, we consider it a subject to future work and concentrate on strictly and unpredictable streams in this work.

Unpredictable streams have the less favourable property that static types are no longer sufficient to express actual dependency. Instead, dependency is defined as a dynamic property of data units. Most of the advanced video coding features in H.264/AVC [27] generate unpredictable dependencies. Examples are adaptive reference pictures, adaptive intra-refresh and variable slice modes. Because of their efficiency and superior error-resilience, unpredictable streams are widely used in lossy packet networks.

As a consequence to this classification, it becomes apparent that object types are not always sufficient to express actual dependency. Hence, we need to model both, static type-based dependency and dynamic data-unit-based dependency. In the remainder of this thesis we call it *object-based dependency* to reflect the well-known relation between types as templates and objects as specific instances of types. The type in unpredictable streams merely defines the set of *potential* dependency relations. Hence, type-based dependency is just the necessary and object-based dependency the sufficient property there.

4.4 Modelling Dependency Relations

The purpose of dependency modelling is to capture data unit relations as well as the unequal and dynamic importance distribution in media streams in real application environments. To reach this goal, we first define the abstract core foundation of our dependency model in mathematical terms in this section. Later, we enhance the core by practical features and high-level operations to deal with special format properties and real environments. For example, in the core model we assume that senders and receivers have perfect knowledge about the past and future of a stream. Later, we introduce tools that assist in predicting invisible structure and dependencies.

4.4.1 Type-based and Object-based Dependency

In our core dependency model we define objects to abstract from data units. Objects are instances of types and the function otype : $O \mapsto T$ obtains the type t of an object o, with O being the set of all objects defined for a particular stream and T being the set of object types defined for a particular stream format. To express dependencies between types and objects we use graphs as mathematical framework. Type-based dependency is typically known prior to stream creation, while object-based dependency is unknown until a stream is actually created. Hence, we define a static graph for type-based dependency, the *type graph* G_T, and a dynamic graph for object-based dependency, the *object graph* G_O.

The type graph is an attributed and directed graph $G_T = (V_T, E_T)$ which may contain parallel edges, loops and cycles. As shown later, we require these properties to express the type relations found in current and future encoding formats. The vertices V_T represent the set of object types $t \in T$ of the particular format and the edges $E_T \subseteq \{V_T \times V_T\}_b$ represent static dependency relations between those types. Note that due to parallel edges, the set E is actually a multiset. Relations in the type graph are uni-directional. They exist between exactly two types. Vertices may contain additional attributes to store extra information used in operations on the type graph, while edges may contain attributes to express relation constraints. Attributes are modelled as edge and vertex labels.

The object graph $G_O = (V_O, E_O)$ is a directed attributed and acyclic graph (DAG). The set of vertices V_O represents the objects $o \in O$ and the set of edges $E_O \subseteq V_O \times V_O$ represents dependency relations between those objects, such that $(v_o, v_{o'}) \in E$ *iff* object o' must be decoded in order to be able to decode object o (in other words: object o depends on o' while object o' is a reference for o). Each vertex can contain additional attributes to express properties not directly related to dependencies. One example is a state attribute that may reflect whether an object is visible or invisible for some reason (e.g. already processed, lost or explicitly dropped).

The type graph is generated once at design time of a bitstream format. Hence every format and most likely every subset of a format is represented by a special type graph. In the design phase, the graph should be validated to detect inconsistencies and to verify uniqueness of types and type relations. At run-time, the type graph remains constant, but it must be available to decorate the object graph. Especially in transmission scenarios, the type graph must be reliably exchanged prior to stream transmission. This may happen at the design-time of a system or during session setup. Costs of exchanging the type graph are negligible because its size is small compared to the typical size of a data unit in video streams.

The object graph, in contrast, can be manipulated at run-time. This may involve adding or deleting vertices and edges as well as modifying their attributes. Depending on the application purpose, the object graph may contain the complete set of objects of a stream or a limited subset only. For transport protocols this may be the set of data units that are in the transmission window, while streaming servers can hold large sections of a stream for quick reaction to playout-state changes. The necessary information to decorate the object graph at run-time is contained (a) in the type graph and (b) in special self-containing meta-data structures, called object labels (see section 4.7.3). Labels may be attached to data units and transferred in transport protocols. When meta-data about objects is unavailable, e.g. due to network loss, type information and dependencies can be recovered from static type info as discussed later.

4.4.2 Dependency-Graph Operations

The core foundation of our dependency model defines a small set of operations to work with type and object graphs. We will later use these simple operations to define more complex functionality. Most of the operations are only meaningful when performed at the object graph because the type graph is static and can be analysed at design time. The basic operations roughly divide into four sets:

Vertex and Edge Generation: Creation and deletion of edges and vertices is limited to the object graph. Hence, we only define the vertex operations

$$\begin{aligned}&\text{create\_vertex} : G_O \mapsto G_O \cup \{v_{o\_new}\} \quad \text{and}\\ &\text{delete\_vertex} : G_O \times V_O \mapsto G_O \setminus \{v_o\}\end{aligned}$$

and the edge operations

$$\begin{aligned}&\text{create\_edge} : G_O \times V_O \times V_O \mapsto G_O \cup \{(v_o, v_{o'})\} \quad \text{and}\\ &\text{delete\_edge} : G_O \times V_O \times V_O \mapsto G_O \setminus \{(v_o, v_{o'})\}.\end{aligned}$$

Vertex deletion implicitly removes all edges originating in this vertex. Removing a vertex that still has inbound edges is not defined.

Vertex Set Selection: Single or multiple vertices in the object graph may be selected based on their attributes, such as for example, their type, state or marks (see section 4.6 for an overview on vertex attributes). For simplicity we model the selection problem with a special template object o_T that is wildcard-matched against all vertices in G_O:

$$\text{find\_vertices} : G_O \times V_O \mapsto \mathcal{P}(V_O).$$

Note that selecting a subset of vertices differs from selecting a subgraph. The resulting set contains vertices only, while a subgraph may contain vertices and edges.

Subgraph Selection: An important property of directed graphs for dependency tracking is the ability to extract connected sub-graphs such as transitive closures and inverse transitive closures for a given vertex. Note that the transitive closure of a root vertex may also be a directed rooted tree (either out-tree or in-tree), but this is no requirement here, because dependency relations do not always form tree-like structures. Closures may be used to perform reachability analysis which is required to check for broken dependency. They are also useful for determining the set of depending objects, which is required to calculate importance metrics or to select drop victims. Our subgraph selection operations are defined as:

$$\begin{aligned}&\text{tc} : G_O \times V_O \mapsto T_O \subset G_O \quad \text{and}\\ &\text{itc} : G_O \times V_O \mapsto S_O \subset G_O.\end{aligned}$$

A transitive closure from an arbitrary vertex is extracted by recursively tracing all vertices reachable via outbound edges. The inverse transitive closure is extracted by inverting the edge directions first. Implementing subgraph selection is straight-forward because no circles and loops are allowed in the object graph.

Attribute Access: Graph attributes may be read or written, whereas the type graph only supports read access. We therefore define set and get operations for object graph vertices and edges as well as get operations for type graph vertices and edges. The attribute sets for type and object graphs are defined in sections 4.5 and 4.6. Here, we specify them as <name, value>-pairs where attribute names are defined by the sets $Attr_{TV}$, $Attr_{TE}$, $Attr_{OV}$, $Attr_{OE}$ for vertex and edge attributes (indices V and E) of type-graphs resp. object-graphs (indices T and O). Valid values for each attribute are defined in separate value sets Val_{Attr_*}. Operations to read object graph attribute are:

$$\begin{aligned}&\text{get\_attribute} : G_O \times O \times Attr_{OV} \mapsto Val_{Attr_{OV}} \quad \text{and}\\ &\text{get\_attribute} : G_O \times O \times O \times Attr_{OE} \mapsto Val_{Attr_{OE}},\end{aligned}$$

operations for object graph attribute write access are:

$$\begin{aligned}&\text{set\_attribute} : G_O \times O \times \mathit{Attr}_{OV} \times \mathit{Val}_{\mathit{Attr}_{OV}} \mapsto \emptyset \quad \text{and}\\ &\text{set\_attribute} : G_O \times O \times O \times \mathit{Attr}_{OE} \times \mathit{Val}_{\mathit{Attr}_{OE}} \mapsto \emptyset,\end{aligned}$$

and operations for read-only type-graph attribute access are:

$$\begin{aligned}&\text{get\_attribute} : G_T \times T \times \mathit{Attr}_{TV} \mapsto \mathit{Val}_{\mathit{Attr}_{TV}} \quad \text{and}\\ &\text{get\_attribute} : G_T \times T \times T \times \mathit{Attr}_{TE} \mapsto \mathit{Val}_{\mathit{Attr}_{TE}}.\end{aligned}$$

4.5 Type-Graph Attributes

A type graph can model all static properties of bitstream formats and their static dependency relations. This is already sufficient to fully describe predictable streams with fixed bitstreams structures, such as MPEG-2 streams used in DVB. Even dynamic bitstreams formats may benefit from static information, because such formats also have several predictable and limitedly predictable features.

The structure of a type graph, for example, already reveals a first approximation of the importance of objects, because types with more dependency relations are likely to produce objects of higher importance. For example, the structures displayed in figure 4.1 have fixed dependencies and the corresponding importance patterns could be generated based on the static type graph.

The graph structure alone is, however, not always sufficient to cover the flexibility of all bitstream formats. Therefore we define additional vertex attributes to (a) express default values, (b) type patterns that can be used for prediction, and (c) additional edge attributes to express dependency constraints. Constraints effectively define the required conditions that must hold true in order to consider a dependency relation between objects valid. These attributes are later used in operations that validate stream integrity and estimate the initial importance of object even when objects itself are lost or dependency relations are hidden.

4.5.1 Dependency Rules

Based on our core dependency model we can already define types as vertices and edges as potential dependency relations between types. In practice, however, not every object of one type does necessarily depend on every other object of a target type. Hence, we need some rules to further confine the relations. To attach rules to edges we already defined edge attributes in the type graph, but before we define expressions for rules, we first take a closer look at bitstream properties.

Please recall that dependency in media streams exists between data units of a specific type, is uni-directional and backward only and it has a limited radius (with the exception of long-term references, we discuss later). Multiple data units may be members of a *group*, such as the fragments of a single frame or the frames in a group of pictures (GOP). Group membership is always exclusive, that is, a data unit is member of at most one group. In general, it is possible that a group has a single member only. Often, bitstreams contain sections of interdependent data units that are decodable without external references. We call such sections *epochs*. In the theory of our core dependency model, an epoch refers to a connected subgraph of the object graph. In practice, a new epoch in video streams starts with an intra-coded frame or an IDR NAL unit which has no dependency to previous sections.

In rare cases, epochs may contain members with strictly limited *external* dependencies. External dependencies are defined, for example, for open-GOPs in MPEG video, where the last B-frame depends on the I-frame of a subsequent GOP. Note, that in this case the transport order of the bitstream also requires GOP *interleaving* to avoid forward dependency. Cross-epoch dependencies unfortunately violate our assumption that epochs are disconnected subgraphs. Hence, we define special edge attributes for type graphs and object graphs to mark these edges. Affected graph operations need to consider the marking.

In certain bitstreams there may exist groups with *internal* relations between members. Internal dependencies originate, for example, from fragmentation, partitioning, layering, FGS or multiple description coding. Common to groups with internal semantics is that all members share the same type and dependency relations to other groups, but the semantics of relations inside the group depends on the type of the group. The semantics may even differ between encoding formats and new semantics are likely to emerge. Therefore we define an extensible set of group semantics in section 4.5.3.

When a dependency relation between two types exists, this indicates that there should also be a dependency between the objects, but sometimes this dependency is meant to be *optional*. For example, headers or parameter sets may be optionally repeated in a stream. When the optional header is missing in the actual bitstream, a following object of a depending type is not necessarily invalid. In contrast, some types are regarded *essential* for the integrity of bitstreams, such as intra-coded reference frames. Essential dependencies are also useful for modelling strictly predictable formats, because all object dependencies must satisfy the type constraints in this case.

In order to statically express the properties we identified so far, the following required and optional constraints may be combined into dependency rules. The rules assume a unique order over all data objects in a stream, which will later be defined by sequence numbers. Each dependency edge in the type graph must be refined by exactly one rule and every rule may contain each of the following options at most once:

Dependency Type: The dependency type defines whether dependencies are optional (`weak`) or essential (`strong`). In parallel type-graph edges, strong and weak dependencies may be mixed, but if the dependency rules are identical in the remaining attributes, the strong relation overrides the weak. The dependency type is a required property for rules.

Distance Constraint: The distance constraint `dist` $\in \mathbb{N}$ defines the size of the dependency radius between origin and destination types of the type-graph edge. The distance is relative to the occurrence of object types in the later object graph rather than an absolute value. A distance value of 2 between a type t_1 and a type t_2, for example, covers a stream section between an occurrence of type t_1 in a stream and the last two occurrences of objects of type t_2. The term last refers to the backward-directed nature of dependencies. The distance is a required property for rules.

Set Operator: In combination with the distance constraint the set operator restricts the set of candidate objects a given object may depend on. The operator may either be `last_of` which selects a single object at the specified distance (e.g. with a distance of 2 it selects the last P-frame prior to another P-frame frame) or `all_of` which likewise selects all objects of the target type within the distance (e.g. it selects the last two P-frames). The set operator is a required property for rules.

Epoch Selection: The epoch selector `epoch` $\in \{0, 1, 2\}$ specifies whether there are any dependencies between the affected types across epochs. It constricts the radius of cross-epoch dependencies relative to the epoch of a later object to the same epoch (`epoch = 0`, no cross-epoch dependencies), the subsequent epoch (`epoch = 1`, useful for open GOPs) or all previous epochs (`epoch = 2`, useful for parameter sets). The epoch selector is optional and defaults to 0.

Epoch Interleaving: The optional `epoch_interleaving` $\in \mathbb{N}_0$ parameter expresses the distance of epoch interleaving between the affected types. When specified, it declares that an origin type object is shifted behind `epoch_interleaving` occurrences of a target type object in the stream although the shifted object is part of the preceding epoch. The default value is 0, which disables interleaving.

Layer Selection: For layered streams, layer selector `layer` $\in \{0, 1\}$ constraints the dependency relation to objects within the same layer (`layer = 0`) or to objects in some lower layer (`layer = 1`). Layer selection should be used when the same object types may appear in more than one layer as is the case for EI, EP and B-slices in H.263++ and H.264/SVC layered encoding. Layer selection is optional.

Figure 4.5 shows an example type graph for the MPEG open-GOP bitstream layout displayed in figure 4.1(a). In this example, I-frames depend on the last sequence header regardless how many epochs it is away. P-frames either depend on a preceding I-frame or a preceding P-frame in the same epoch, whichever is closer. A B-frame depends on at least two other frames, a preceding I-frame in the current or the subsequent epoch, or one or two preceding P-frames, if they are within the B-frame's epoch. The B-frame which depends on a subsequent epoch's I-frame is also shifted behind the occurrence of the I-frame in stream transport order. This effectively expresses the observable epoch interleaving.

In grouped streams the distance-specific operations consider all members of a group as single objects. When frames are fragmented into multiple objects, for example, the fragments are likely to have the same type. Then, object-graph operations need to consider multiple group members as single objects in order to avoid wrong application of dependency rules.

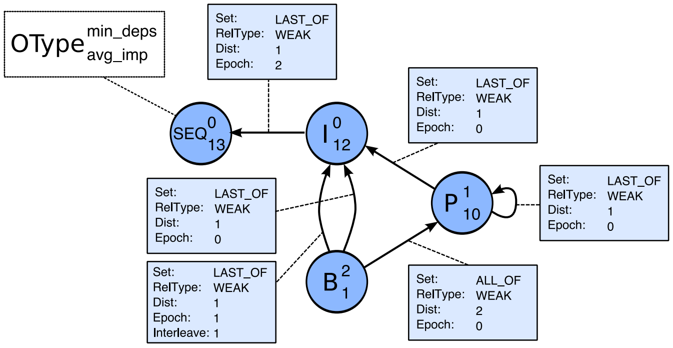


Fig. 4.5 : *Type-graph for MPEG-like streams with I-P-B pattern.*

4.5.2 Type Attributes

The type graph is meant to assist dependency tracking, validation and estimation operations with static properties that are known at design time of a bitstream format. Our core dependency model already allows us to attach arbitrary attributes to vertices and edges in the type graph. Here, we discuss some attributes we discovered while analysing different bitstream formats. One set of attributes helps us later when decorating the object graph and a second set provides support for predicting structure and importance if possible for this stream format.

Limits. When generating and validating the object graph later, we need information about the number of expected and required dependency relations for every object type. Although the actual number of dependencies may differ between objects and across formats, there is always a maximal and a minimal limit for practical reasons. In fixed-dependency formats upper and lower limits are equal. Dynamic formats, in contrast, often require a minimal number of dependency relations to satisfy decoding dependencies at all, while an upper bound of possible relations is defined per format or per stream. Fixed-type examples are B-frames in MPEG streams that require exactly two surrounding reference frames to be decodable. Dynamic formats like H.264/AVC allow multiple and weighted reference pictures. They usually define an upper limit to restrict decoder buffers and a lower limit (often a single reference picture) to exploit inter-frame correlation at all.

To express such limits, we define the type attribute `min_deps` $\in \mathbb{N}_0$ to express a minimal number of required dependencies and the attribute `max_deps` $\in \mathbb{N}_0$ to express a maximal number of desired dependencies. `min_deps` is optional and it defaults to the number of strong relations defined for this type or 1 if only weak relations exist. An object of this type is considered valid (its necessary dependencies are satisfied) if at least `min_deps` relations to other valid objects exist and all strong dependencies are satisfied. Otherwise the object has a broken dependency. `max_deps` merely defines a stop criterion that is used when inserting new edges into the object graph without the availability of explicit references. It is also optional and it defaults to the value set for `min_deps`.

Prediction. Hidden dependency relations and loss decreases the importance prediction accuracy of our model. In section 4.6.2 we will introduce operations that can predict certain properties of streams. Here we define an initial set of type attributes that enable the implementation of simple prediction algorithms. These attributes are optional. They are only meaningful for predictable object types. Future extensions of the dependency model may define additional attributes or support more sophisticated algorithms.

For importance prediction we define the type attribute `avg_imp` $\in \mathbb{N}$. A format designer can use it to specify a minimal importance value that is used for all objects of this type as long as the dependency-based importance is not higher. Different objects of this type may have a real importance that is actually lower or higher than this value. Therefore a format designer should choose a reasonable average that balances between the negative effects of overestimation and underestimation.

For prediction purposes we define three type attributes that describe the recurring structural patterns of a type: `period` $\in \mathbb{N}$, `offset` $\in \mathbb{N}_0$, and `burst` $\in \mathbb{N}$. Period specifies how frequent the pattern for this type repeats. The offset specifies at which sequence number the first period starts and the burst attribute defines how many successive data units of the specified type are expected per period. Different types may have different periods. They are not required to share a common period, such as the GOP length. In addition to the pattern we also define the type attribute `starts_epoch` $\in \{true, false\}$ to mark the object type that starts new epochs. At most a single type in the whole type definition of a stream format may be selected. If there is no exclusive type candidate, the attribute should be omitted.

4.5.3 Group Semantics

We define the optional type attribute `group_semantic` $\in \{none, equal, unequal, refinement\}$ to express the semantics of object groups. This selection reflects the distinct types of group semantics we identified so far: (1) equal containment groups, (2) unequal containment groups, and (3) refinement groups. The default value for the group semantics is *none*.

In *equal containment groups*, members share the same importance, while the group is only valid if all members are available. This is similar to fragmented data packets and useful to model data units that are fragmented due to network restrictions. *Unequal containment groups* contain members with different importance values, relative to their group importance. We require group members to be ordered by decreasing importance. Such a group is valid if all members with importance offset larger than the value of the optional type attribute `min_imp` $\in \mathbb{N}_0$ are available. This semantics is applicable for data partitioning and unequal error protection, where a single application-level data unit is split into unequally important fragments. Finally, the members of a *refinement group* become more important when less members are available. At least a single object is required to make a refinement group valid.

4.6 Object-Graph Attributes and Operations

The dynamic properties of objects that go beyond the static information of the type graph are expressed by attributes attached to vertices in the object graph. There are four different sets of vertex attributes to describe the identity, type, and structural relations of data objects as well as additional parameters. The set of attributes is open for later extensions which may be required by new algorithms that operate on the object graph. The current algorithms work on the following meta-data:

Sequence Number: The sequence number expresses the identity of a data object. They must be natural numbers that are greater than 0 and they should increase monotonically with every data object in the stream. Sequence number must be unique for every instance of a stream. Uniqueness requires that every data object has a single sequence number only and every sequence number is only used for a single object.

Type: The type attribute specifies the type of a data unit. This attribute must contain one of the values specified in the type graph. The type attribute is necessary for unpredictable streams, where the type cannot be inferred from the sequence number. In such streams, it serves as the only connection between the object graph and the type graph.

Epoch: Epochs express to which independent stream section the object belongs. For a stream instance they should be unique monotonically increasing natural numbers. Within each connected subgraph of the object graph this attribute must be equal. It is used to detect boundaries of subgraphs more easily when cross-epoch dependency is allowed by a stream format. Epochs may also be used for garbage collection of expired information.

Group Information: While sequence numbers already define identity, group membership and position inside a group is expressed with two group attributes `group_seq` and `group_size`. `group_seq` specifies the position of the data object in the group which starts at 1. `group_size` specifies the overall number of group members which is at least 1. For all members the group size is equal. Group members are also required to occupy a contiguous sequence number space. We use this redundant scheme to enable robust identification of groups even if members are reordered and lost. Without groups, both attributes must be set to 1.

Layer Information: Layered encoding defines additional dimensions of dependencies that may be orthogonal to the normal dependency relations. Layering usually forms refinement hierarchies between multiple data objects. To express this extra dimension, we specify two attributes which are defined as natural numbers. The `encoding_layer` attribute defines the layer number the object belongs to and the `referenced_layer` attribute defines the hierarchically closest layer that is used as a reference. Both attributes may be used in addition to dependency relations, while the exact semantics of the values may be redefined in special algorithms. For H.264/SVC streams, for example, the values can be re-interpreted as hierarchical operation points for downscaling and even as multi-dimensional or multi-purpose values (e.g. to express multiple scaling dimensions simultaneously).

State: The `state` attribute is an application-specific extension that may reflect whether a stream object is available, was already processed, lost or explicitly dropped. When selecting vertex sets or subgraphs or when validating for broken dependencies object-graph operations use this information. The state values have a fixed and pre-determined semantics for all operations. The state must be explicitly set by an application.

Long-Term Reference Flag: Some coding standards like H.264/AVC define long-term reference data objects. Long-term objects are likely to have inbound cross-epoch dependencies and they are not subject to epoch-based garbage collection. To reflect this special properties and to provide an anchor for later implementation optimisations, we define the `long_term` flag as attribute.

Marked Flag: Applications can set and remove the `mark` flag in order to explicitly include or exclude marked objects from object graph operations. This may be used by an application to efficiently simulate "what if" conditions without changing state or object graph structure. The marked flag may also be set or removed by advanced dependency-graph operations to store a simple state across multiple chained operations. In contrast to the state attribute, marks have no common pre-determined semantics between all dependency-graph operations.

Importance Offset: The importance offset attribute `imp_offset` is an optional integer attribute that specifies an object-specific importance value. Offset values are used to increase robustness in general and expressiveness in intra-group relations. For extra-ordinary important data objects such as long-term references and parameter sets the offset can reflect the real importance even if dependency chains are broken due to loss. For grouped streams, the offset is interpreted depending on the group semantics. It may decrease, limit, or increase the estimated importance of a data object in addition to the dependency-related importance. For equal groups the offset defines a maximal limit for the importance value, for unequal groups it defines an additional importance to be added or subtracted from the group importance, and for refinement groups it specifies a basic importance value that increases when group members are lost.

Practically, the uniqueness property of sequence numbers requires that one instance of the object graph must only be used for a single stream in a single session. The algorithms presented in this chapter strictly assume these properties. We will later use sequence numbers for ordering, (backward-)referencing, and loss detection in the stream sequence.

When reusing sequence numbers from other subsystems, care is required. While encoders, streaming servers and transport protocols already use some kind of sequence numbering scheme, these sequence numbers may not fulfil the requirements stated above. Problems usually arise when numbers from a different abstraction level or time-frame are used. Assume, for example, a streaming server that uses the dependency model to select data units for playout to clients. The server may send a particular data object multiple times during a session when playout direction and order are altered by a user. While the sequence numbers the server uses for a stream remain constant, a lower layer streaming protocol (that may also use an instance of our dependency model for selective error control) will assign new sequence numbers to every transported object. Hence, both sequence schemes are incompatible.

4.6.1 Object Graph Decoration

While the type graph is statically generated at design time, the object graph is generated at run-time, either once when a session is established or in regular intervals as new or retransmitted data objects become visible. We call the process of object-graph generation *graph decoration.* It uses the dependency relations contained in the type graph as well as meta-data about data objects such as their type and explicit references. The type graph is necessary to provide implicit dependencies for predictable types and to verify the explicit dependencies against the dependency constraints.

The decoration operation $\mathrm{decorate}(G_O, G_T, o)$ adds the vertex v_o and eventually new adjacent edges to the object graph G_O using the operations defined in section 4.4.2. Vertex attributes are set according to the meta-information provided for object o. An edge $(v_o, v_{o'})$ is added if one of the two following conditions is true:

1. o' satisfies a dependency relation, defined in the type graph G_T for $\mathrm{otype}(o)$ when the object label's reference list is empty (*implicit decoration*),

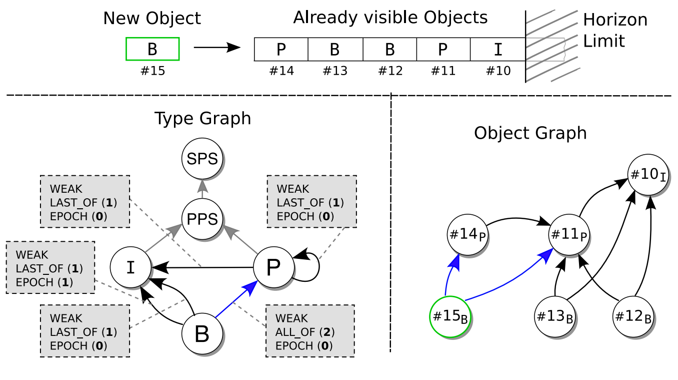

New Object
B
#15
Already visible Objects
P
B
B
P
I
#14
#13
#12
#11
#10
Horizon Limit
Type Graph
SPS
PPS
I
P
B
WEAK LAST_OF (1) EPOCH (0)
WEAK LAST_OF (1) EPOCH (0)
WEAK LAST_OF (1) EPOCH (1)
WEAK LAST_OF (1) EPOCH (0)
WEAK ALL_OF (2) EPOCH (0)
Object Graph
#10I
#14P
#11P
#15B
#13B
#12B

Fig. 4.6 : *Implicit decoration, based on type information only: A new object of type B and with sequence number 15 is inserted into the object graph. The actual edges to existing vertices are drawn based on the type-graph relation between type B and P. This is because vertex 14 and vertex 11 are the closest vertices to 15 and the B -> P relation matches. The max_deps value for type B is 2 (not shown in the type-graph) and decoration stops after inserting two edges.*

2. o' is contained in the object's reference list and a dependency relation between otype(o) and otype(o') in G_T is satisfied (*explicit decoration*).

If a stream contains groups, no edges between group members are required in the object graph since the group is already described by the `seq`, `group_size` and `group_seq` attributes. References across groups always point to the first member objects which serves as proxy into its group.

For efficiency reasons, an implementation may choose to mark the newly decorated object to have a broken dependency if a lost object, an explicitly dropped object or any transitive reference of such an object is selected as reference, or a required group member is missing. Broken dependencies may also be detected by transitively inspecting the object graph and checking for group members later. Figure 4.6 shows the implicit decoration of data unit 15, a B-frame. For this type a maximum of 2 references is required. Both are satisfied by the same dependency relation between B- and P-type and hence, edges to preceding objects of type P are inserted into the object graph.

A problem arises when data units are invisible and types are unpredictable because the information from the type graph is no longer sufficient and implicit decoration fails. Figure 4.7(a) displays such a situation where the loss of unit 14 causes implicit decoration to wrongly choose data unit 10 as reference instead of detecting a broken dependency.

This problem can either be resolved by type prediction or by explicit decoration as shown in figure 4.7(b). While prediction is limited to certain stream formats, explicit dependency lists work for all stream classes. Especially behind loss-gaps the decoration algorithm must ensure that subsequent valid objects are decorated correctly, so that no invalid relations are inserted into the graph. This property can only be ensured if explicit references are available. Thus, explicit reference lists are essential for robustness. Note that the loss of explicit reference lists can only lead to an incomplete object graph, but never an incorrect one.

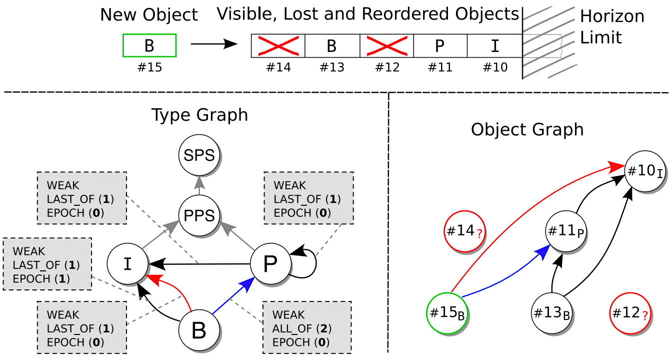
When storage requirements of the graph are important for an application, a garbage collection algorithm should be used to prune old data. Garbage collection must consider short-term and long-term objects separately. When new data objects from new epochs become visible, old short-term references may become unimportant, while long-term references remain essential until explicitly removed.

4.6.2 Structure and Importance Prediction

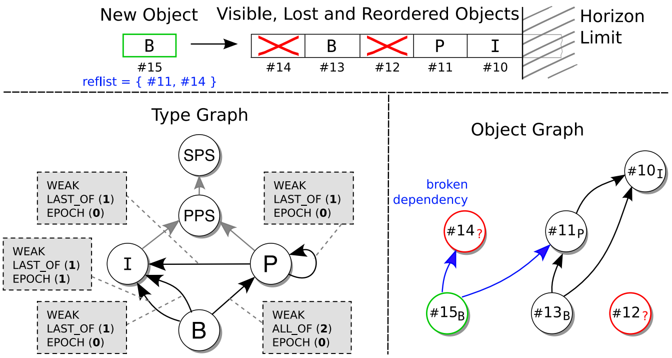
Resilience to loss and reordering of data units requires the prediction of importance and dependency. This section presents algorithms to recover essential meta-data attributes, such as type, epoch, and potential dependency relations of invisible data objects from sequence numbers and type attributes. Fortunately, current encoding standards define bitstream profiles that allow such predictions, and real applications use these schemes to avoid implementation complexity.

Because the design space for predictability is large, we select some important prediction examples only. The proposed techniques assume a well formed periodic structure, which is only found in strictly predictable and limitedly predictable media streams. They are not generally applicable to adaptive and unpredictable stream formats. We consider this future work.

Predicting Importance from Types It is sometimes unknown how many data objects will later depend on an object when it becomes initially visible and hence, how important this object finally becomes. If this information was not provided in the label attribute `imp_offset` or if the


New Object
Visible, Lost and Reordered Objects
Horizon Limit
B
#15
B
P
I
#14
#13
#12
#11
#10
Type Graph
Object Graph
SPS
PPS
I
P
B
WEAK LAST_OF (1) EPOCH (0)
WEAK LAST_OF (1) EPOCH (0)
WEAK LAST_OF (1) EPOCH (1)
WEAK LAST_OF (1) EPOCH (0)
WEAK ALL_OF (2) EPOCH (0)
#10I
#14?
#11P
#15B
#13B
#12?

(a) Loss limits the applicability of implicit decoration.


New Object
Visible, Lost and Reordered Objects
Horizon Limit
B
#15
reflist = { #11, #14 }
B
P
I
#14
#13
#12
#11
#10
Type Graph
Object Graph
SPS
PPS
I
P
B
WEAK LAST_OF (1) EPOCH (0)
WEAK LAST_OF (1) EPOCH (0)
WEAK LAST_OF (1) EPOCH (1)
WEAK LAST_OF (1) EPOCH (0)
WEAK ALL_OF (2) EPOCH (0)
broken dependency
#10I
#14?
#11P
#15B
#13B
#12?

(b) Explicit decoration increases the resilience against loss.

Fig. 4.7 : *Limits of implicit decoration and benefits of explicit decoration: When loss leads to unavailable type information, implicit decoration may choose the wrong target objects which leads to improper graph structures and undetected broken dependencies. Explict decoration ensures at least a valid structure of the graph, althought the graph may be incomplete.*

data unit is lost and one likes to infer the severity of the loss, the type attribute `avg_imp` can be used as a good approximation. When we only know the sequence number, as is the case for lost and reordered objects we need to predict the type first.

Predicting Object Types from Sequence Numbers Efficient type prediction (without explicitly describing the stream layout) is only possible if a stream features periodically recurring patterns. Alternatively to prediction we could exchange traces of the future stream pattern in advance, but this requires extra resources and is infeasible in most cases. Instead, we use a small set of attributes to describe patterns in general. These attributes are only useful for strictly predictable streams because they are fixed for a whole sequence.

The following pseudo code shows a simple prediction algorithm that makes use of the prediction attributes defined in section 4.5.2. For reasons of clarity and brevity we ignore variable sized fragment groups and groups in general.

Algorithm 1 Type prediction algorithm for fixed stream structures.

```
Input: seq
Output: type
1: theType $\Leftarrow$ invalid
2: for all types t $\in$ T do
3:    if ((seq - offset(t)) mod period(t) $\leq$ burst(t) &&
      (avg_imp(theType) < avg_imp(t))) then
4:       theType $\Leftarrow$ t
5:    end if
6: end for
7: return theType
```

This algorithm checks the specified sequence number against all defined type patterns. If multiple types match, the most important one is selected. Hence, a lost data unit may be regarded more important, but never regarded less important as it really is. This helps to avoid mistreatment in content-aware protocols and creates equally important loss gaps in the worst case. The algorithm can also predict the group membership of lost objects. This becomes valuable if all group members are lost and neither size nor the start of a group are derivable (`group_size` and `group_seq` are lost too). As with all predictions, this requires a regular structure with fixed group sizes over the total sequence.

Most of today's encoding standards define profiles with fixed bitstream patterns (e.g. MPEG-2 or H.264/AVC over Digital Video Broadcast channels). For these classes of applications it is always possible to disclose the types of invisible and lost data units. Limitedly predictable streams must, however, use at least periodic updates during the lifetime of a session to inform about upcoming pattern changes.

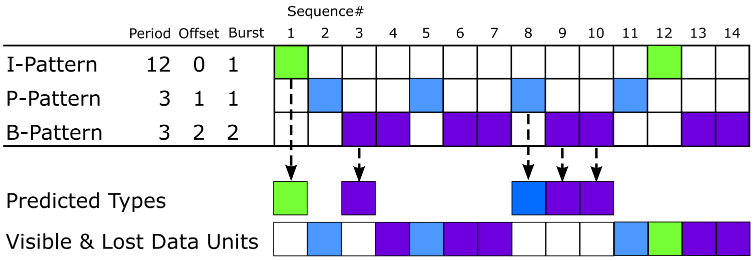
For example, assume the structure of MPEG streams with open-GOP pattern: (1) GOP(12,2) with a distance of 12 frames between two I-frames and two successive B-frames between I/P-frames, and (2) GOP(16, 15) with temporal scalability, having a distance of 16 frames between two I-frames and 15 successive B-frames in four temporal layers, but lacks P-frames. The definition of the prediction values for the open GOP pattern (12,2) is:

	First GOP			Remaining GOPs		
Type	**Period**	**Offset**	**Burst**	**Period**	**Offset**	**Burst**
I-Frame	12	0	1	13	10	1
P-Frame	3	1	1	3	13	1
B-Frame	3	2	2	3	11	2

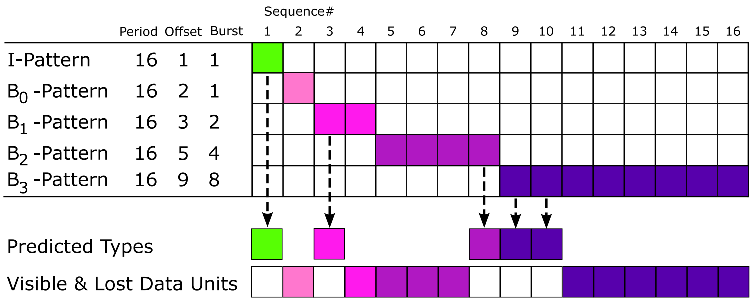
Figure 4.8 depicts how the prediction algorithm matches lost data units with their correct types. Note that the algorithm always assumes a fixed and repeatable pattern per type.

Predicting Epochs from Sequences and Types Epoch values are required for correct implicit object-graph decoration and the removing of old data units. To infer the epoch values of lost data objects we use a simple scheme that relies on the already predicted type property. From the type graph we know the type attribute `starts_epoch` that marks the type which starts a new epoch. However, the actual epoch must still be calculated from the sequence number. The epoch size can be inferred from the `period` value that was specified for the epoch start type. The actual epoch value is the modulo of the sequence to the epoch size.

When a stream format interleaves adjacent epochs, such as in the MPEG open-GOP structure, epoch calculation must also account for the interleaving between types. Therefore we already defined the relation attribute `interleaving` for all relations that point to the epoch start type. This interleaving specifies how many object of the origin type follow an object of the epoch start type (in sequence number order), but come from the previous epoch. Based on this information, the epoch prediction algorithm works as follows:



(a) GOP(12,2)



(b) GOP(16,15)

Fig. 4.8 : *Type prediction example: when the structure of a stream is well formed, it becomes possible to predict the type of lost data units; the example shows the first group-of-pictures of two popular structures, found in MPEG-2 and MPEG-4 video streams.*

Predicting Object Types from known References Incoming references from other objects reflect the importance of the referenced data unit, even if the unit itself is missing. It is obvious to use in-bound references for type prediction as well, since the number and the type of referencing objects yields some hints on the type of the invisible object. They can be compared to the average importance, the minimal and the maximal dependency count attributes in the type graph to find an estimate of the type in question. This method requires at least some other data objects and their explicit references to be present. It is inappropriate for prediction in large gaps generated by burst loss, but it is a superior method for unpredictable streams.

Once the type of a lost data object can be recovered with an acceptable probability, dependency relations in the type graph can be used to

Algorithm 2 Epoch prediction algorithm that accounts for epoch interleaving.

```
Input: seq, epoch_size, interleaving
Output: epoch
1: theEpoch $\Leftarrow$ (seq $\div$ epoch_size) + 1
2: if (0 < (seq  mod  epoch_size) $\leq$ interleaving) then
3:     theEpoch $\Leftarrow$ theEpoch - 1
4: end if
5: return theEpoch
```

implicitly decorate the object graph. More sophisticated prediction algorithms are possible. However, we regard the presented mechanisms as sufficient and defer extensions to future work.

We can conclude that for strictly and limitedly predictable streams, type information and eventually all dependency relations may be recovered. This is, however, impossible for unpredictable streams. They require explicit reference lists to recover at lease some metadata of referenced objects. Because reference lists are also subject to loss, our model become less accurate at higher loss rates. With short epochs in the stream our model can still recover quickly.

4.6.3 Dependency Validation

The purpose of dependency validation is to ensure that all necessary (object-based) and sufficient (type-based) dependency relations for a given object and its transitive predecessors are satisfied so that the object may be successfully decoded. Figure 4.9 shows a schematic example.

Dependency validation is implemented by the function *valid*: $O \mapsto \{true, false\}$, using the object graph and the type graph. The object graph is prepared for validation during decoration and hence the *valid* operation is able to efficiently check the following four conditions: A data object $o \in O$ is valid if all of the following conditions are satisfied:

(a) it is neither dropped nor lost itself,

(b) it has no broken dependency,

(c) all its strong dependency relations are satisfied, and

(d) its minimum dependency `min_deps` is satisfied.

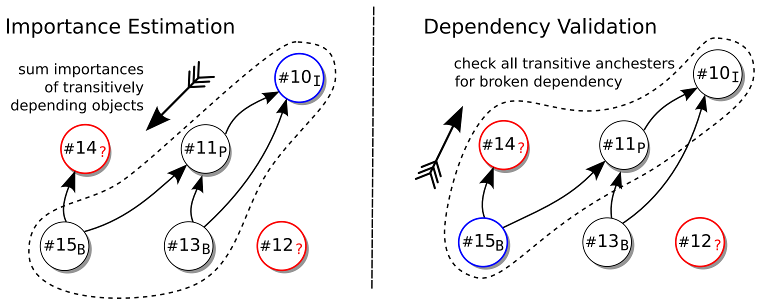

Importance Estimation
sum importances of transitively depending objects
#10I
#14?
#11P
#15B
#13B
#12?
Dependency Validation
check all transitive anchesters for broken dependency
#10I
#14?
#11P
#15B
#13B
#12?

Fig. 4.9 : *Importance estimation vs. dependency validation processes: While estimations uses the transitive closure of a target object to calculate the number of depending objects, validation must ensure that no object in the inverse transitive closure experienced a broken dependency.*

When one of the group semantics *equal* or *unequal* is defined for the type of the data object o, additional properties must hold:

(e) if the object is member of an equal containment group it is valid *iff* all members of its group are valid too, or

(f) if the object is member of an unequal containment group it is valid if and only if all group members with smaller group sequence number than object o are available *and* all group members with importance offset value `imp_offset` larger or equal than the required minimal group importance, specified in type property `min_imp`, are available. (Note: both ranges overlap, but they may be equal if object o is exactly the last required group member.)

Especially rule (f) is complicated to check under loss because `imp_offset` is unknown for lost objects. Therefore we require the object attribute `imp_offset` to decrease monotonically with increasing group sequence number when the unequal group semantics is applied.

Objects of types with group semantics *none* and *refinement* are not affected by additional rules since they are either not members of any group or they do not depend on the existence of their group members. When using the model for strictly predictable streams, dependency validation is even possible in the case of lost or corrupted objects. This is not because the dependency relations of the lost objects can be reconstructed, which is unimportant anyway. It becomes possible because the object type is predictable and decoration as well as validation of subsequent objects can rely on this information. When streams are unpredictable, the type of lost objects is not recoverable and type-based dependency alone becomes insufficient. Then, explicit object reference lists are required to compensate for the lack of information.

4.6.4 Importance Estimation

The importance of an object depends on the information encoded within the object itself as well as on the importance and number of objects which depend on this information directly or transitively. The dependency model only captures the transitive dependency-related importance and leaves other properties to additional models, such as object-specific rate-distortion or timeout values. Note that in general the dependency-based importance is a relative value between visible data objects. This is because the model cannot know the actual importance and dependencies of invisible objects and it cannot predict lost offset values. Importance values between different media streams are also not comparable.

Due to the differences between simple and group-based importance estimation, we propose different mechanisms, all based on the core operations defined in section 4.4.2. We start with defining the operation for simple importance estimation. As with validation, the importance function $\mathrm{imp}_{\mathrm{nogroup}} : O \mapsto \mathbb{N}$ uses information from both graphs to estimate the actual importance value. If no group semantics is defined for the type of object o, the importance of object o is the maximum of its type importance and its meta-importance, additionally increased by the correction value stored in the object's label:

$$\begin{aligned}\mathrm{imp}_{\mathrm{nogroup}}(o) := \; & \max\Big\{\mathrm{type\_imp}\big(\mathrm{otype}(o)\big),\ \mathrm{meta\_imp}(o)\Big\} \\ & + \mathrm{imp\_offset}(o).\end{aligned} \tag{4.1}$$

The *meta-importance* reflects the maximal depth of any path in the transitive closure of object o in G_O. The recursive definition ensures that meta-importance values are monotonically increasing with the length of dependency chains between objects.

$$\mathrm{meta\_imp}(o) := \max_{o' \,\in\, \mathrm{tc}(G_O, o)} \Big\{\mathrm{meta\_imp}(o')\Big\} + 1. \tag{4.2}$$

The type importance for type t is defined to be the maximum of (a) the type-specific `avg_imp` value, (b) the number of vertices in the transitive closure of t in G_T increased by 1 or (c) the maximum of `avg_imp` over all vertices in the transitive closure increased by 1:

$$\begin{aligned} \text{type\_imp}(t) := \max \Big\{ & \text{avg\_imp}(t), \\ & \big|\,\text{tc}(G_T, t)\,\big| + 1, \\ & \max_{t_i \,\in\, \text{tc}(G_T,t)} \big\{ \text{avg\_imp}(t_i) \big\} + 1 \Big\}. \end{aligned} \tag{4.3}$$

When a group semantics is defined for an object type, the calculation is performed differently. First, a basic group importance value is determined using the function *group_imp*. It is based on the dependency of the group head object with group sequence 1 (one). Group heads represent the group regardless whether they are lost or available themselves. When all visible group members are lost, neither a correct group membership is deducible nor a correct estimation of loss severity is possible. When the group size is constant and predictable (see section 4.6.2) the type prediction algorithm can compensate for the missing information. Unpredictable streams, however, suffer from wrong estimations because only lost group heads collect references and other lost members remain at their initial importance value of one (1).

Group importance is similar to object meta-importance, except that the importance offset is handled differently to capture the relative increase when group members define importance offset values themselves. The group importance is defined to be the maximum of the group importance values of all objects in the transitive closure (tc) of the group head o_1 in G_O increased by the maximum of the importance offset values for all group members (we neglect the definition of the function *group* $\mapsto \mathcal{P}(O)$ for brevity.):

$$\begin{aligned} \text{group\_imp}(o) := & \max_{\substack{o_1 \,\in\, \text{group}(o) \,\wedge \\ \text{group\_seq}(o_1) = 1 \,\wedge \\ o' \,\in\, \text{tc}(G_O, o_1)}} \Big\{ \text{group\_imp}(o') \Big\} + \\ & \max_{o_m \,\in\, \text{group}(o)} \Big\{ \text{imp\_offset}(o_m) \Big\} + 1. \end{aligned} \tag{4.4}$$

The recursion in the function *group_imp* and the inclusion of group importance offset values is necessary to ensure that no object in any depending group can become more important than its parent objects, even if it specifies a large importance offset. While in general the absolute importance values are larger in group-based streams when compared to streams without groups, the relative importance between objects of a type remains stable. Depending on the group semantics, the importance estimation for group members (including the group heads) is performed as follows.

Equal containment groups define the importance of a member as the maximum of the member's type importance and the common group importance value:

$$\begin{aligned}\mathrm{imp}_{\mathrm{equalgroup}}(o) := \max\Big\{\, \mathrm{type\_imp}\big(\mathrm{otype}(o)\big),\ \mathrm{group\_imp}(o)\Big\} \\ +\ \mathrm{imp\_offset}(o).\end{aligned} \tag{4.5}$$

The equality constraint forbids the use of an additional member-specific importance offset because it is already included in the group importance. The loss of any member may lower the absolute group importance due to the potential lack of a new maximum offset, but it does not influence the relative inter-group relationships. Lost members itself loose their private offset values however. Hence, all members of equal groups should have the same offset to increase the loss resilience of importance estimation.

Unequal containment groups define the importance of members relative to the group importance in the range:

$$\Big[\mathrm{group\_imp}(o) - \max\Big\{\, \mathrm{imp\_offset}(o_m)\Big\},\ \mathrm{group\_imp}(o)\Big]_{o_m\, \in\, \mathrm{group}(o)}. \tag{4.6}$$

For every object o the group importance is decreased by the difference between the maximal importance offset defined by any group member and the private importance offset of object o:

$$\begin{aligned}\mathrm{imp}_{\mathrm{unequalgroup}}(o) := \mathrm{group\_imp}(o) - \max_{o_m\, \in\, \mathrm{group}(o)}\Big\{\, \mathrm{imp\_offset}(o_m)\Big\} \\ +\ \mathrm{imp\_offset}(o).\end{aligned} \tag{4.7}$$

The loss of a group member does not influence the importance values of other members because the potentially lost maximum is neutralised by the relative difference in the equation. The importance of a lost member is, however, only deducible from its position in the group. As already required for correct validation, the offset values for unequal groups should therefore be monotonically decreasing with increasing group sequences when loss resilience is of primary interest.

Refinement groups define the importance of members relative to the number of unavailable group members. Hence, the fewer objects of a refinement group are available the higher the importance values of remaining members and gaps become. Refinement groups are somewhat special because importance is allowed to become larger than the importance of referenced objects. Hence the importance maximum is not bound by the properties defined for the function *group_imp*. Instead, the importance of an object o in a refinement group is the object meta-importance of the group head o_1, increased by the number of missing group members and additionally increased by the private offset value of object o:

$$\begin{aligned} \mathrm{imp}_{\mathrm{refinementgroup}}(o) := & \max_{\substack{o_1 \in \mathrm{group}(o) \wedge \\ \mathrm{group\_seq}(o_1) = 1}} \Big\{ \mathrm{type\_imp}\big(\mathrm{otype}(o)\big),\ \mathrm{meta\_imp}(o_1) \Big\} \\ & + \mathrm{group\_size}(o) - \Big| \ \mathrm{group}(o) \Big| + \mathrm{imp\_offset}(o). \end{aligned} \tag{4.8}$$

When offset values throughout the stream are carefully designed, the relative importance between incomplete refinement groups and referenced groups (or single objects) can be balanced appropriately. Refinement-based importance cannot raise above referenced object importance when the referenced objects possess an average type importance larger than the largest group size of any refinement group or a similar importance offset is set on members basis. Because any loss to a refinement group member makes its offset value and the actual group size unavailable, it is a good choice to leave the default of 0 (zero) for importance offset when loss robustness is desired. Additionally, the missing group size of lost objects can be obtained from available group members.

Discussion Unlike validation which uses backward dependency, importance estimation relies on forward dependency. Figure 4.9 shows the concept of the importance estimation process. Estimation is performed for any member of the object graph individually. Type prediction and explicit dependencies of objects ensure that even lost objects can be referenced. When loss is irreparable, broken dependency chains may severely influence the correctness of estimation as we will see in section 4.8.3. As a short example, consider the loss of P-frame 14 in figure 4.9. While the lost frame remains (transitively) referenced by subsequent P-frames, the I-frame 10 is disconnected due to the lack of a reference from 14. In consequence, the importance of frame 10 is wrongly estimated even though explicit dependency lists are in use. Type prediction and implicit decoration can compensate for this kind of problem, but unfortunately they are unavailable when the stream format is generally unpredictable. This is the price of unpredictability.

Exact importance values for every object can only be calculated for strictly predictable streams. In unpredictable streams exact importance is only available when all transitively depending objects are visible within the horizon. It is impossible to provide exact importance values if some depending objects are still invisible or lost. In section 4.8.2 we discuss this limitation in more detail. Consequently, an initially estimation of object importance is based on the object type. It is re-examined when more dependencies become visible or loss is repaired.

4.7 Dependency Model Implementation

Every stream format uses its private syntax elements and dependency relations between them. A generic dependency model implementation is therefore required to provide a suitable abstraction for programmers to map a specific encoding to types and objects.

In order to package and describe bitstreams, we provide programmers with the abstraction of *labelled data units*. Fragmentation and labelling are left to the application programmers, but we provide them with tools to ease these tasks. For format designers who need to specify types and type-relations we define a special *Dependency Description Language* (DDL) and the corresponding DDL-Compiler. The DDL Compiler verifies the language description and translate it into a compressed representation of the type graph.

The format designer is required to select a packetisation scheme where the desired dependency relations are visible. Common packetisation schemes operate, for example, at the frame-, slice- or network-packet level. A single format can even be described at different abstraction levels for different applications. An application developer should select an appropriate scheme and partition the bitstream accordingly.

We implemented the run-time version of our dependency framework as a self-contained *Dependency Validation Service* (DVS) that can be integrated into applications and system layers. The DVS only caches information about data units without requiring data units to be present. It therefore introduces no conceptual delay to stream processing. The DVS may be used in servers, proxies and client applications, for live and on-demand streaming, and in typical packet networks where data units are reordered and lost. Applications can control the amount of storage used by the DVS by selecting a garbage collection strategy to fit the model's horizon to their window of interest. Garbage collection removes information about old and processed data units.

In the following sections we first present the dependency description language, provide some examples on how to use this language for real-world video formats and finally describe the API and implementation of the dependency validation service.

4.7.1 Dependency Description Language

In order to specify types, type-based dependency relations and dependency constraints we developed a dependency description language (DDL). The DDL compiler verifies the validity and consistency of the description and transforms it into a compact string representation which required to initialise the DVS at run-time. The compact representation can be integrated statically into applications, embedded into media signalling protocols and file headers, or transported by other means.

The language provides constructs to express structure and properties of the type graph, such as vertices (the types), edges (the dependency relations), as well as vertex and edge attributes. In order to support strictly predictable and unpredictable formats, some of the attributes are optional. The full syntax description and the extended Backus-Naur form of DDL is in appendix A.

A DDL description contains two sections. First, all types are declared and their parameters are defined. Next, the desired relations, including necessary constraints, are defined, whereas dependency definition is

```
# DDL Example for an I-P-B-like Dependency Pattern
#
types = {
    seq_head( avg_imp = 255, min_deps = 0 );
    i_frame( avg_imp = 12, min_deps = 1, max_deps = 1 );
    p_frame( avg_imp = 2, min_deps = 1, max_deps = 1 );
    b_frame( avg_imp = 1, min_deps = 2, max_deps = 2 );
}

dependency(i_frame) = {
    last_of(seq_head) weak;
}

dependency(p_frame) = {
    last_of(i_frame) weak;
    last_of(p_frame) weak;
}

dependency(b_frame) = {
    last_of(i_frame, 1, 0) weak; # I-frame in the same epoch
    last_of(i_frame, 1, 1) weak; # I-frame in the subsequent epoch
    all_of(p_frame, 2, 0) weak; # P-frame(s) in the same epoch
}
```

Listing 4.1: *Dependency description for a MPEG-like video stream.*

performed for each origin type separately. Listing 4.1 shows an example on how to express the typical I-P-B-P frame dependency pattern of predictively coded MPEG video streams as depicted in figure 4.5. For illustration purposes, this example displays a simplified structure between the most well known bitstream elements only. Realistic streams contain more types and relations, but are not necessarily more complex as we will show in section 4.7.2 when describing the H.264/AVC video format.

This hypothetical stream format contains four types. The sequence head carries the sequence start code and parameter sets to configure the decoder. It is the first unit in a sequence and also the most important unit. Hence it does not depend on other units and it has a large average importance. All other types depend on the sequence head because it's presence is essential for decoding the sequence at all. We neglect relations from every other type for brevity and because the `i_frame` type already establishes transitivity. The `i_frame` type only depends on the sequence head and is self containing otherwise. Assuming a GOP size of 12, the average importance is set to the largest possible dependency chain length, which occurs when no B-frames are used. A P-frame depends on either a directly preceding I-frame or P-frame, whichever comes first. This is reflected by two weak last_of dependencies, both with a default distance of one. Because a P-frame always requires a single reference the type-specific attributes min_deps and max_deps are set to one. The B-frame dependency is more complex because there are several alternatives. B-frames either depend on two P-frames or on one I-frame and one P-frame, whereas the I-frame may belong to the current or the subsequent GOP. In order to express the possible variations neither of the dependencies can be strong. Validity requires, however, at least two references (expressed by $min\_deps = 2$) and it makes no sense to have more than two references (hence $max\_deps = 2$).

Language Constructs The **types keyword** declares a list of types for a particular format and hence the set of vertices if the type graph. In addition, it allows the format designer to specify a set of required and optional attributes, which are attached to the graph as vertex labels. Although attributes are already discussed in sections 4.5.2 and 4.6.2, we briefly recap their meaning in the context of the DDL:

avg_imp (required) defines an average estimate on the type's importance. This estimate is used for importance prediction when a data unit is lost, when dependency relations are yet invisible, or when a combination of both applies. Valid values are positive integer numbers except zero.

min_deps (required) defines the minimal number of outgoing references a data unit of the respective type must have in order to become valid. (An outgoing reference is a dependency relation that has a particular data unit as origin.) Valid values for min_deps are positive integer numbers including zero.

max_deps (optional) defines the maximal number of references a data unit of the respective type is required to have. This value determines when to stop decorating the object-graph. If not provided or set to the default value zero, all matching units in the horizon will be referenced. Valid values are positive integer numbers including zero; the default is zero.

min_imp (optional) defines the importance limit above which unequal containment groups are considered valid when some group members are still missing. In particular, the limit requires that all group members with an importance offset larger than min_imp are available. This attribute is only used for unequal containment groups.

Valid values are positive integer numbers including zero, while a value of zero means that all group members must be available; the default is zero.

prediction_period, prediction_offset, and prediction_burst (optional) define the recurring pattern of data units of a particular type. The attributes are only used for type prediction and setting them makes only sense for predictable streams. Valid values are integers including zero. Offset is defined relative to the period, starting at zero. Burst determines the number of consecutive occurrences of a type. The size of both values is limited by the period length, which can be the GOP size. A period of zero or a burst length of zero disables prediction for the respective type. It is neither necessary to define the same period for all types of a stream, nor is it necessary to define these parameters for every type, although it is recommended.

group_semantic (optional) defines how the importance for data units of the respective type is calculated (see section 4.6.4). Valid values are `none`, `equal`, `unequal`, and `refinement`, the default is `none`.

starts_epoch (optional) defines whether the respective type starts a new epoch. This attribute is required for correct epoch prediction, but it is an exclusive property that must be assigned to at most one type. Format designers should only used this feature when a particular type does always start a new epoch and if no other type starts an epoch throughout the stream. Valid values are `true` and `false`; the default is `false`.

The **dependency keyword** declares a block of potential dependency relations that originate in a particular type and have other types or the same type as target. The semantics of the dependency styles for set selection (last_of, all_of) and relation kind (weak, strong) are those defined in section 4.5.1. In DDL we use the set selector as principal entity that starts a relation, the kind as a mandatory modifier that closes a relation, and the remaining attributes as optional parameters. Each declaration ends with a semicolon. This form makes a relation naturally readable and unambiguous.

The origin type for a relation is specified as a parameter to the dependency keyword, while the particular target type is specified as the first parameter to the set selector. In order to use a type as origin or target it must have been defined. Dependency constraints are optional parameters to a set selector. They are determined by their position in the argument list. In the order of occurrence the have the following meaning and defaults (in brackets): distance (1), epoch distance (0), layer distance (0), epoch interleaving (0). When defaults are sufficient, these parameters can be omitted. When, however, a parameter at the end of the list should be specified, all preceding parameters must be specified too. A relation ends with its kind, either essential (strong) or optional (weak).

DDL Compiler The purpose of the DDL compiler is to validate and compress the type-graph, initially specified in the dependency description language. Therefore the compiler checks the syntax of the description, validates the semantics of the type-graph and transforms the textual representation into a target representation which is later required to initialise an instance of the dependency validation service. Currently, we use a compact bracket-based ASCII-string representation as output format. The benefits of this format are its platform independence, human readability and size. It can be statically embedded into application code, stored in file headers and transferred via media signalling protocols.

The DDL compiler ensures that there are no cycles, no conflicting relations, and no invalid or contradicting parameters in the graph. The graph contains plausible parameters if:

- only a single type defines the starts_epoch attribute
- $\forall \, types \, t : \; max\_deps(t) >= min\_deps(t)$
- $\forall \, typest : \; prediction\_offset + prediction\_burst <= prediction\_period.$

Two edges are in conflict if

- both have the same origin and the same destination, and
- both have the same distance, epoch distance, and layer distance, and
- both have the same kind (strong or weak).

The cycle check ensures that no type and thus no data unit in the object graph has a direct or indirect self reference. While the type graph may contain loops and cycles they are required to be distinct in at least one of the layer distance or epoch distance constraints.

4.7.2 H.264/AVC Video Stream Example

The following examples shows how the DDL can be used to express dependency relations in H.264/AVC [27,49], a predictive and block-based video stream format. We motivate the design considerations we made to map the characteristics of H.264/AVC onto a DDL specification. The example starts with a brief discussion of relevant concepts and features of H.264/AVC. The terms picture and frame are used interchangeably herein.

The central unit of operation in H.264/AVC is a slice, a consecutive number of macroblocks in a picture. H.264/AVC defines two slice partitioning modes, one mode where each picture is fragmented into a fixed number of slices (there can also be as less as a single slice per picture) and another mode where the size of slices is controlled to not exceed a specified threshold, which is usually set to the network MTU size. While the first mode generates variable sized slices according to the prediction and macroblock encoding modes used, the second mode generates a variable number of variable sized slices. Each slice contains the ID of the picture it belongs to. The slice-type determines which prediction modes are used for the contained macroblocks. As in traditional MPEG streams, there are I-slices for self-containing intra-coded data, P-slices and B-slices for predicted data, but also the new types SI-slices and SP-slices for switching between resolutions. Dependency is expressed in reference lists contained in each slice header. Predicted slices always depend on one or more reference pictures. The distance from a reference picture can be limited for a particular sequence (e.g. to 16 pictures), but it may also be longer when a reference picture is marked as long-term reference. H.264/AVC supports several (reference picture-)memory management commands (MMCO) used to designate reference pictures to long-term references and back. In order to combine information about encoder profile, colour formats, picture numbering, coding modes and other features which apply to multiple sequences or multiple pictures of a sequence H.264/AVC introduces sequence and picture parameter sets. Parameter sets are essential for decoding a sequence and hence, they are the most important units in a stream.

In order to respect the properties of different distribution channels, H.264 introduces a Network Adaptation Layer (NAL) which defines several transport unit types (see table 4.1). Each slice is transported in a single NAL unit, whereas a one-byte NAL unit header determines the unit type and whether the contained data is used as reference. This

Type	Content
0	Unspecified
1	Coded slice of a non-IDR picture
2	Coded slice data partition A
3	Coded slice data partition B
4	Coded slice data partition C
5	Coded slice of an IDR picture
6	Supplemental enhancement information (SEI)
7	Sequence parameter set
8	Picture parameter set
9	Access unit delimiter
10	End of sequence
11	End of stream
12	Filler data
13	Sequence parameter set extension
14	Prefix NAL unit in scalable extension
15	Subset sequence parameter set
16..18	Reserved
19	Coded slice of an auxiliary coded picture without partitioning
20	Coded slice of in scalable extension
21..23	Reserved
24..31	Unspecified

Tab. 4.1 : *NAL unit types in H.264/AVC and H.264/SVC.*

wrapping of slices, however, hides some information about slices such as their real type and dependency. When data partitioning is used, the data bits of a single non-IDR slice are distributed over three NAL units, a partition A that contains all header bits, a partition B that contains intra-coded data and a partition C that contains predicted data.

In order to fully capture the elements in H.264/AVC transport streams in our dependency model, it is necessary to model DDL-types at the NAL unit level, because only here all relevant concepts (parameter sets, partitions, etc.) are visible. Unfortunately, NAL unit types do not reflect the actual slice types. As a further disadvantage, the flexible slice modes make H.264/AVC streams generally unpredictable. When, however, the number of slices per frame is fixed and the dependency pattern is fixed too (e.g. I-P-B-P), a H.264/AVC sequence becomes predictable. This comes at the cost of reduced coding efficiency and reduced error resilience.

Since our model requires its own unique sequence numbering scheme we do not rely on H.264/AVC picture order counts. We do also ignore memory management commands of H.264/AVC because they add a considerable amount of complexity and they are not error resilient. Rather than fixing these issues we provide a simpler scheme to directly mark long-term reference units in unit labels. Once marked, a unit remains a long-term reference until the marking is removed.

Listing 4.2 displays the most relevant parts of the H.264/AVC dependency description (a detailed version is in appendix A, figure A.2). Because the parameter sets SPS and PPS are the most essential data units they have a very large average importance. The PPS essentially depends on a previous SPS even if the last SPS is several epochs away. Hence, the dependency relation is strong and value 2 was chosen as the epoch distance. All NAL units that contain frame data strongly depend on both parameter sets, regardless of the epoch in which the last parameter set was transmitted.

IDR and non-IDR NAL unit types that carry frame data are the most frequent types in simple profile sequences. Both have at least two strong references to the latest SPS and PPS. Non-IDR units can have considerable more references, but they can also contain intra-coded data without dependency. Hence we set their minimal dependency to two even if they may have more dependency relations besides the strong dependency from parameter sets, but other dependency relations are optional (weak). In addition we define an equal group semantics because slice partitioning may generate multiple NAL units per frame. Although we assume equal groups with equally important fragments here, an unequal importance distribution is still implementable via offset values in labels.

In data partitioning mode, non-IDR data is transported in data partition NAL units. These units form an unequal containment group per frame. The unequal importance of partition types is reflected in the average values and the selected group semantics. Because this is only an initial estimate, the offset values of data partition units must be set too.

The remaining NAL unit types carry extra data that has no references and that is not referenced. Hence the minimal dependency is zero and no dependency declaration is specified. Note that the average importance values for all types reflect the relative initial importance only. When more data units become visible, actual dependency relations quickly overrule these small values. The unpredictable nature of general H.264 streams requires that most dependency information is contained in data unit labels. The presented type system merely serves as integrity check and as a source for initial importance values.

```
#  DDL Description for General H.264/AVC Sequences (partial)
types = {
  # Parameter Sets
  NALU_7_SPS(avg_imp = 255, min_deps = 0);
  NALU_8_PPS(avg_imp = 253, min_deps = 1);

  # Frame-data Containers
  NALU_5_IDR(avg_imp = 6, min_deps = 2, group_semantic = equal);
  NALU_1_NON_IDR(avg_imp = 5, min_deps = 2, group_semantic = equal);

  # Data Partitioning
  NALU_2_DPA(avg_imp = 5, min_deps = 2, group_semantic = unequal);
  NALU_3_DPB(avg_imp = 4, min_deps = 3, group_semantic = unequal);
  NALU_4_DPC(avg_imp = 3, min_deps = 3, group_semantic = unequal);

  # Extra Data Containers
  NALU_6_SEI(avg_imp = 1, min_deps = 0);
  NALU_9_AUD(avg_imp = 2, min_deps = 0);
  NALU_10_EOSQ(avg_imp = 2, min_deps = 0);
  NALU_11_EOS(avg_imp = 2, min_deps = 0);
  NALU_12_FILL(avg_imp = 2, min_deps = 0);
}

dependency(NALU_8_PPS) = {
  last_of(NALU_7_SPS, 1, 2) strong;
}

dependency(NALU_5_IDR) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_8_PPS, 1, 2) strong;
}

dependency(NALU_1_NON_IDR) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_1_NON_IDR) weak;
}

dependency(NALU_2_DPA) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_2_DPA) weak;
}

dependency(NALU_3_DPB) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_2_DPA) weak;
}

dependency(NALU_4_DPC) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_2_DPA) weak;
}
```

Listing 4.2: *H.264/AVC dependency description (some types are stripped for brevity, see listing A.2 in appendix A for the full description).*

4.7.3 Dependency Validation Service

The Dependency Validation Service (DVS) implements an instance of the dependency model and offers interfaces for manipulation, validation and estimation. It enables applications and streaming protocol services such as scheduling, error protection and content scaling to access dependency and importance values while these services can remain unaware of the actual data unit content otherwise.

The DVS only requires labels as input data, it does not assume to see the payload of data units. Hence it may run detached from the actual data path. When a stream flows, the DVS should be notified about new data units, detected sequence gaps, and data units that have been processed, dropped or successfully retransmitted. The DVS keeps track of data unit states and updates the corresponding graph nodes. Streaming servers that have full knowledge about a stream can load sections and even the total stream description into the DVS, depending on the available memory resources.

The interface of the DVS is organised into four sections: decoration, validation, marking and estimation (see listing 4.3), which correspond to the dependency model functions *decorate*, *valid* and *imp*.

In general, the DVS performs the following tasks:

- find transitive dependants of a given data unit (the data unit's transitive closure in the object graph),
- find units, transitively referenced by a given data unit (the data unit's inverse transitive closure in the object graph),
- determine if a unit is invalidated by a previous loss or drop,
- estimate the importance of visible and lost data units, and
- estimate the conditional importance of data units given a constraint.

Data unit sequence numbers are used as a common index to data units and internal graph-based representations. For decoration, the DVS requires that sequence numbers are presented in monotonically increasing order. Gaps are considered as loss and according vertexes are inserted into the internal graph. All other methods that operate on sequence numbers require that numbers are in the range of the current horizon. An error is raised when an index is out of bounds.

```
// Dependency Validation Service -- API
//
enum {AVAIL, LOST, DROPPED, PROCESSED} State_t;
enum {KeepValid, MaxHorizon, MaxEpoch} GCMode_t;
enum {NoPred, ImpPred, TypePred} PredictionMode_t;
enum {ImplicitDeco, ExplicitDeco} DecoMode_t;

struct Rating_t {
    int     seq, imp, type_id;
    bool    valid, marked, long_term;
    State_t state;
};

struct Label_t; // see table 4.2

// Initialisation
DVS(string typeDesc, int maxSize, int maxLongTerm,
    GCMode_t gcMode, PredictionMode_t predMode,
    DecoMode_t decoMode);

// Graph Decoration
void insertObject(Label_t label);
void insertLostObject(int seq);
void updateObject(Label_t label);
void updateObjectState(int seq, State_t newState, bool mark);

// Structure Validation
bool isValid(int seq);
State_t getState(int seq);
RatingVector_t getDependants(int seq);
RatingVector_t getAncestors(int seq);

// Data Unit Marking
void setMark(int seq);
void clearMark(int seq);
bool isMarked(int seq);

// Importance Estimation
int getImportance(int seq);
int getConditionalImportance(int seq, bool cond);
Rating_t getRating(int seq);
RatingVector_t getMostImportant(State_t state, int howMany,
                                bool invertState);
RatingVector_t getAllByImportance(State_t state, bool invertState);
```

Listing 4.3: *Dependency Validation Service API.*

Initialisation A DVS instance is initialised using the dependency description as generated by the DDL compiler, the maximal horizon size and the maximal number of long-term references for horizon size control and to estimate the memory requirements, and several operation-mode selections. Negotiation and transport of dependency descriptions and the identification of correct values for `maxSize` and `maxLongTerm` are be-

Purpose	Attribute	Description
Strictly predictable streams	`seq`	unique sequence number ($\in T$)
Limitedly predictable streams	`type`	data unit type
	`epoch`	dependency epoch ($\in \mathbb{N}$)
	`enclayer`	encoding layer (optional) ($\in \mathbb{N}_0$)
	`reflayer`	referenced layer (optional) ($\in \mathbb{N}_0$)
Unpredictable streams	`short_term_reflist`	seq. of short-term references
	`long_term_reflist`	seq. of long-term references
	`is_long_term_ref`	flag to mark as long-term reference
Group-based streams	`group_seq`	in-group position ($\in \mathbb{N}$)
	`group_size`	number of data units in this group ($\in \mathbb{N}$)
	`imp_offset`	additional importance offset ($\in \mathbb{N}_0$)

Tab. 4.2 : *Dependency-Related attributes in data unit labels.*

yond the scope of the service. Maximum values can be obtained from analysing a particular stream or from the encoder setup. The operation modes control how decoration and prediction are performed. They allow the customisation of the DVS for strictly predictable streams and unpredictable streams. The decoration mode determines if the dependency graph is to be decorated using implicit type-based knowledge only or if decoration should rely on explicit reference lists in labels. The prediction mode controls if the importance is predicted from types (ImpPred), if importance and types are predicted from sequence numbers when possible (TypePred), or if prediction is disabled.

Data Unit Labels The properties of each data unit are specified by a common set of meta-data we call *label* (see table 4.2). A label must contain all information required to decorate the object graph and set the desired vertex attributes. The information in labels must follow the requirements of vertex attributes defined in section 4.6.

Labels may be attached to data units during transport and processing or may be transported and stored detached. The DVS only assumes that they are properly generated and inserted before operations on the represented sequence numbers are allowed. In addition to the already defined object attributes, a label may contain *explicit reference lists.* We define two reference lists, one for short-term reference units within the dependency radius of the data unit (usually the current epoch) and one for more distant long-term reference units. If present, a list contains the sequence numbers of explicitly referenced data units. If absent, type-based relations from the type graph are used for decoration instead. When a data unit is part of a group, the reference lists should contain the sequence numbers of group heads only. All information in labels can be easily derived from encoders and bitstream parsers. For efficient access by streaming servers it should be stored in hint tracks.

Labels also define the non-static properties of a group membership for each data unit. Recall, that the group semantics is already defined in type attributes. For each data unit, `group_seq` specifies the position of the data unit within the group (starting with one (1)), and `group_size` specifies the number of group members (at least one (1)). Group members are required to have contiguous sequence numbers. This enables robust identification of the first element in each group using its sequence number and the group sequence number, even if the head of a group is missing due to reordering or loss. When the group feature is unused, `group_seq` and `group_size` must be set to one (1).

Data Unit Lifecycle When new data units become visible or gaps in sequence numbers are detected, `insertObject()` or `insertLostObject()` should be called to notify the DVS. Both methods decorate the object graph appropriately. If a reordered or retransmitted data unit arrives later, the method `updateObject()` should be used. It takes special care of properly updating the object graph. When a data unit is processed or dropped `updateObjectState()` tells the dependency model about this event. Note that there is no special remove operation, because the DVS manages the horizon based on a garbage collection strategy.

Horizon Control (Garbage Collection) Horizon control ensures that information about old and unused data units is finally removed from the DVS. The DVS provides three modes to satisfy different requirements. The *KeepValid*-Mode only removes graph nodes when they are no longer transitively referenced by any node of a still available data unit. The availability can be controlled by the state attribute. In this case, the `maxSize` parameter is unused. This strategy provides the strongest guarantees about availability, but it defers state and memory management to the application. In contrary the *MaxHorizon*-Mode enforces a hard limit on the horizon's maximal size and thus the storage requirement. Graph nodes are removed in strict FIFO order when the maximal horizon size is reached, regardless of state and dependencies. The `maxSize` parameter controls the maximal permitted number of data units in the horizon. Finally, the *MaxEpoch*-Mode limits the horizon size to at most `maxSize` epochs and removes all graph-nodes outside this boundary regardless of their state and dependencies. Because the epoch size may be dynamic for a stream, storage requirements are unpredictable but limited.

Long-term references are handled specially because they remain valid over long time-frames. They are subject to an own garbage collection cycle that uses a LRU (least-recently used) replacement strategy. The number of permitted long-term data units is specified by the `maxLongTerm` initialisation parameter.

Data Unit Marking The DVS provides methods to set, remove and check for marks as a general tool to attach a simple application-specific state to data units. In combination with marks, the DVS offers the concept of *conditional importance* through a second estimation method. When calculating importance estimations, conditional importance only considers data units that have been marked (`cond = true`) or not marked (`cond = false`). This is useful to make simulations without altering the state of data units and without changing the horizon.

As an example, consider deadline control. Intentionally, neither the dependency model nor the DVS know time or the deadlines of data units. Streaming protocols, however, consider time and importance together. Moreover, protocols may wish to relate passed deadlines to a decrease in importance. Assume, for example, a protocol wants to transfer data units when they are already behind their deadline, but some depending units can still reach their deadline. Then late units are still important for decoding until the last depending unit passed its deadline. The marking feature can be used to mark all data units that already passed the deadline. As long as dependency relations to unmarked data units exists, the conditional importance is greater than zero, but it decreases with increasing number of marks.

Validation and Dependency Reasoning For the purpose of validating and reasoning about the actual dependency pattern between visible data units the DVS provides methods to check for broken dependency (`isValid()`), obtain a list of depending data units (`getDependants()`) and a list of referenced units (`getAncestors()`), ask for an importance-ordered list of the n most important units (`getMostImportant`) and an importance-ordered list of units in a specific state (`getAllByImportance`).

The methods return rating structures which contain the dynamic state of each data unit as seen by the dependency model. Content-aware scheduling algorithms, error-control protocols and scaling schemes can directly use this information to perform more educated decisions.

4.7.4 Embedding the DVS into System Layers

Figure 4.10 shows how the content-awareness framework can be embedded into a transport protocol layer. The necessary changes to make a system layer content-aware are minimal because the DVS hides most dependency tracking issues transparently. Required changes are an extension to system interfaces to pass a label in combination with each data unit and to exchange the static type description once at session setup. Internally, a content-aware system layer forwards the labels to the DVS, while content-unaware system layers either pass labels as opaque data or simply discard them. When scheduling, error- and flow-control schemes require information about data unit validity and importance, they are obtained from the DVS.

An application performs processing and fragmentation of streams into data units as usual. In addition, the application must also provide the static type description and generate the data unit labels. The type description for a particular format is specified once at the design-time of the format by a format designer. It can be easily exchanged as description file. A critical part is, however, the extraction of dependency relations because the content-awareness framework provides no format-specific support. We expect network-adaptive media encoders and media file formats to provide proper interfaces in the future. In our prototype

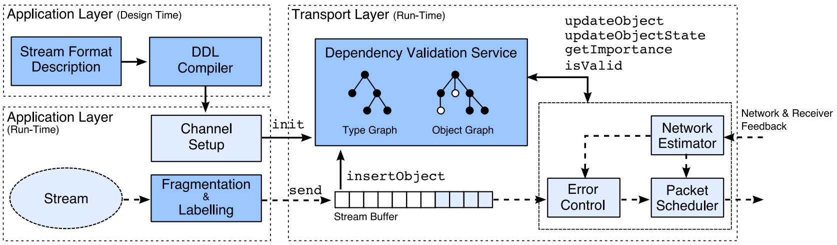

Application Layer (Design Time)
Stream Format Description
DDL Compiler
Application Layer (Run-Time)
Channel Setup
Stream
Fragmentation & Labelling
Transport Layer (Run-Time)
Dependency Validation Service
Type Graph
Object Graph
updateObject
updateObjectState
getImportance
isValid
init
send
insertObject
Stream Buffer
Network Estimator
Error Control
Packet Scheduler
Network & Receiver Feedback

Fig. 4.10 : *Cross-Layer design using the content-awareness framework.*

system we extract information from H.264 elementary streams with our own bitstream parser.

For transporting labels across networks, most of the information already fits into protocols such as RTP and its extension headers while the static format description can be exchanged with media signalling protocols. Only the dependency information requires special extension or payload headers.

4.8 Limitations of the Dependency Model

This section explores several problems in practical systems that may lead to imprecise estimations of the importance of data units. We will discuss the source of the problems and give solutions on how to circumvent them or lower their negative impact.

4.8.1 Estimation Accuracy

As motivated in section 4.3.1, dependency-based importance is less accurate than traditional distortion metrics such as the pixel-based MSE distortion. This is because we only extract some properties of a bitstream for dependency evaluation. Distortion metrics, in contrast, consider the reconstructed image quality and its deterioration due to loss. In addition, our dependency model treats fragmented data units as equally important because all fragments contribute to the same frame. The actual distortion, however, depends on content-specific factors, such as the amount of lost motion and detail information.

Hence we expect a quantitative mismatch between dependency-based importance and distortion metrics. This is already visible in the value ranges. While MSE distortion, for example, uses real numbers to express luminance differences of reconstructed pixels, the dependency model uses integers to count the maximal depth of dependency chains.

Figure 4.11 illustrates the differences for the Foreman sequence (see also section 4.9), which has a considerable amount of local and global motion. The sequence was encoded with x264, an open-source H.264 encoder, at a bitrate of 1000 kbit/s with a GOP-size of 48 and GOP-pattern I-P-B-P. The upper figure shows the calculated distortion of the luminance component (Y) for each frame in the reconstructed sequence after a particular frame was removed to simulate isolated loss. This is a common practise to calculate distortion values [162]. The lower fig-

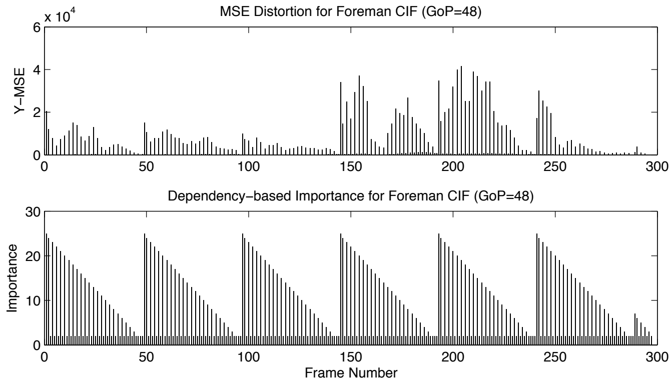


Fig. 4.11 : *Comparison between MSE distortion and importance distributions in the Foreman sequence, encoded with X.264 (one NAL unit per frame).*

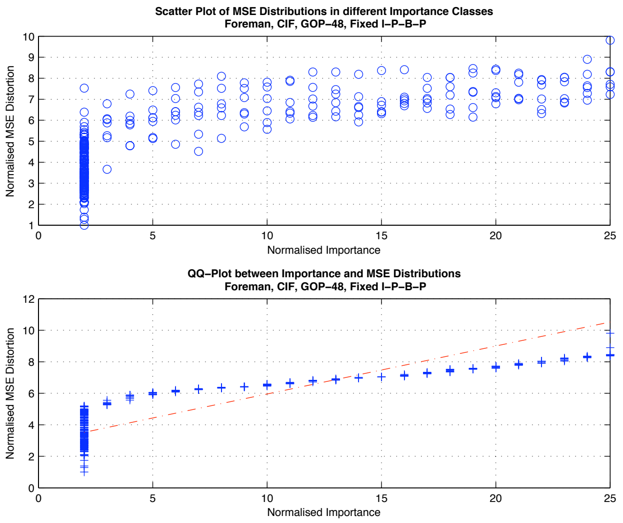


Fig. 4.12 : *Scatter and Quantile-Quantile Plots between MSE distortion and importance distributions in Foreman; points close to the read line indicate a linear relationship between between both data sets; Spearman rank-order correlation coefficient is 0.861, Pearson product-moment correlation coefficient = 0.797, outlier ratio = 5.7 %.*

ure shows the corresponding importance distribution, obtained from the dependency relations using our model. Both figures clearly show that the importance of B-frames is negligible. Quantitative and also a small amount of qualitative differences, mostly between frames 150 and 250, are visible. The scatter- and quantile-quantile-plots in figure 4.12 reveal more details about the actual fit between both models. The dependency model perfectly estimates importance values in the centre of both distributions. Very small and very large distortion values are less precisely approximated. The correlation between distortion and importance becomes non-linear at the distribution tails. The general tendency of the Scatter-Plot in combination with the high rank-order correlation coefficient suggests that dependency-based importance can in fact reproduce the qualitative importance relation between data units. The remaining inaccuracy cannot be eliminated by dependency modelling. It will remain as an artefact of the less precise input information a dependency model uses. In section 4.9 we will more closely examine factors that contribute to inaccuracy and hence to poor importance estimation for a broad number of video sequences.

4.8.2 Horizon Size and Visibility

The dependency model does not assume total knowledge about the importance distribution or the dependency pattern of a particular stream. It only requires information in the type graph and in data unit labels. As mentioned in section 4.3.3 the visibility of data units in a stream is often restricted by application constraints and storage limits. Hence, dynamic information may be invisible and this can considerably limit the importance estimation accuracy.

Figure 4.13(a) depicts how limited visibility affects the actual importance distribution in video sequences over time. The figure take a closer look at the evolution of importance (plotted as colour) for a fixed set of data units (plotted over the x-axis). As time elapses (normalised per data unit and plotted over the y-axis bottom to top) more data units become visible and hence the importance of already visible data units increases until it approaches a final value. The width of the colour stripes depicts the size of each video frame in data units (here H.264 NAL units). Both sequences have a regular structure of 7 unequally sized data units per frame.

The time until an importance value becomes stable heavily depends on the number of subsequent dependencies, e.g. the NAL units of the

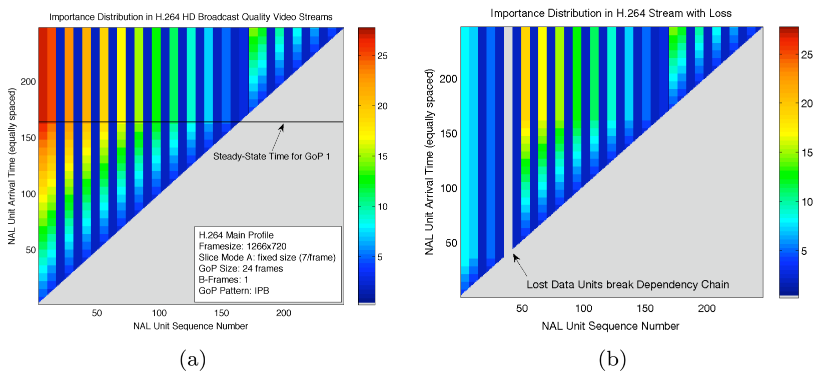


Fig. 4.13 : *Importance distribution maps for H.264 video streams without (left) and with the effects of loss (right).*

I-frame are much longer unstable than late P-frame units. B-frame units (visible as blue stripes) are not affected at all. Importance values of a sequence only saturate when all dependencies are visible, that is, when all data units from a GOP are visible. In the example, two group of pictures are displayed. In the first GOP all data units are already visible, while in the second GOP only the leading I-frame is visible yet. For data units in the first GOP all transitive dependency relations are already present and hence their importance values are saturated. We call such data units to have reached *steady-state*.

Definition 4.8.1 (Steady-state Importance:) *steady-state is reached if all transitive dependency relations for a data unit are visible.*

Steady-state allows to calculate the final importance of a data unit. The single I-frame in the second GOP, however, currently lacks all its dependency relation; it has not yet reached steady state. A naive importance estimation algorithm that accounts for visible dependency only, would consider the single I-frame less important than most of the P-frames in the first GOP. This is clearly a violation of the guarantee the dependency model aims to give. We call this effect *Importance Inversion*:

Definition 4.8.2 (Importance Inversion:) *When for any two non-lost data units u_i and u_j, where at least one unit has not yet reached* *steady state, the* order *of their importance values differs from* steady-state order*, the importance of* u_i *and* u_j *is inverted.*

Importance inversion may seriously affect the usefulness of scheduling decisions based on dependency-related importance. Therefore inversions must be avoided either by the model itself, or by a conscious use of the model and by careful consideration of the values it emits. As we will discuss shortly, this is possible for predictable streams and carefully designed type-graphs, but hard for unpredictable streams. Because inversions are impossible in steady state it is always a safe advice to rely on estimated values for steady-state data units only. Waiting for steady-state is, however, problematic in low delay applications. The steady-state problem is generally decidable when data units are processed in sequence order, because steady-state for a data unit is reached when the first data unit of any subsequent epoch becomes visible. This is because we defined that epochs limit dependency relations. The steady-state problem is even decidable for corrupted streams where some data units are missing because either a subsequent data unit with an epoch value larger than a previous value will be eventually received or the stream ends. Even if reordering occurs, steady-state is decidable when the acceptable out-of-order radius is considered too.

As figure 4.13(a) indicates, for more important data units it takes a considerable amount of time to approach steady-state. The first I-frame remains unsteady until all other 167 data units of the first GOP are visible, while B-frames and the last P-frame immediately enter steady-state at their initial importance. Even if a stream is unpredictable, it has the important property that data units approach steady-state in transmission order.

Another form of invisibility can affect the importance estimation even more severly. As a second example, figure 4.13(b) shows the effects of loss on the evolution of dependency values. When loss leads to broken dependency chains, the dependency model can no longer calculate all dependency values correctly. In this example, the importance of I-frame units who's vertexes are disconnected from subsequent units remains low, while following P-frames still have transitive dependencies. Obviously, the actual importance values display a wrong situation. The result of this obervations is that the estimation accuracy of our model suffers from such forms of invisibility because our algorithms operate on visible data units and visible dependency relations only.

The dependency model already offers ways to recover unavailable information. We first focus on simple solutions for regular and limitedly predictable streams that assume no gaps in the sequence, while section 4.8.3 discusses some advanced solutions.

Prediction assumes fixed regular stream structures and dependency pattern. This assumption may appear restrictive, but predictable streams do not suffer from limited visibility because their actual dependency pattern is inferred from the type graph. For predictable streams it is even irrelevant whether they arrive in-order, because the only information required for prediction is the sequence number (see section 4.6.2).

Average importance values can be specified for every type in the type graph. The importance estimation algorithms use these values as a lower bound for data units of the particular type. Average importance requires a type system that actually reflects importance and dependency. The H.264 simple profile, as a counter-example, supports only two transport unit types for frame content, IDR NAL units and non-IDR NAL units. Given the fact that non-IDR units can contain arbitrary slice types with arbitrary dependency relations, assigning average values at this abstraction level is impossible. If, however, characteristic values for some or all types in a stream format can be identified, the initial estimate can approximate the steady-state values with reasonable precision. This is always possible for strictly predictable streams and it may be possible for limitedly predictable streams too.

Importance offset values can be attached to data units to reflect an increase or decrease of importance in addition to the type averages and dependency-based values. On a per-data unit basis the importance estimation can be fine-tuned by the application, but it is also possible to assign very large or very low importance values without risking a higher probability of importance inversion. Note, however, that offset values are always added to the calculated importance for every data unit. They can easily overrule dependency-based values. In addition, when every unit gets an offset value relative to the unit's steady-state importance, the relative importance order of all units remains unchanged. Hence, it is advisable to use offsets selectively for prominent data units only.

Horizon size control is a technique that requires cooperation from the users of the dependency model. The main idea is to adaptively resize the horizon of the model to hold multiple dependency epochs at once and to rely on estimated values only when the data units have reached steady-state. This is when the first data unit of a subsequent epoch (e.g. group-of-pictures) becomes visible. This technique introduces a pre-roll delay of one epoch and it requires a sufficient amount of storage for data units. It does therefore only work for delay-insensitive streaming applications such as broadcast and on-demand services. Horizon control does not assume that the maximal epoch size (in data units) is known in advance, but implementations will become more efficient when this value is fixed, at least at run-time.

4.8.3 Loss-Resilience

While a stream sender usually has complete knowledge about a stream at least up to the visibility horizon, receivers (and proxies) are affected by loss and reordering of data units. The only reliable information at the receiving side are the static type description and labels of correctly received data units. For lost units, only sequence numbers are known, but types and explicit dependencies are missing.

When linking data units are lost such as depicted in figure 4.14, the missing dependency information can result in fragmented dependency chains and in the worst case in importance inversion. The example shows the effects of a single loss to a simple MPEG-like dependency structure. In this case an important unit in the middle of the dependency chain is lost. The chain is fragmented into two disconnected partitions whereas the first P-frame in the second partition remains reachable by subsequent units, but the more important frames in the first partition become unreachable. Consequently, the transitive importance from the second partition cannot propagate to the first partition, resulting in inverted importance values.

Since inversion can lead to incorrect scheduling decisions or sub-optimal error protection and thus degraded performance, it should be avoided. Average importance and offset values can only provide limited help and, moreover, they apply to available or predictable data units only and they do not work for group-based streams. In order to become more resilient to isolated and bursty loss we suggest additional solutions:

Protocol support and especially importance-based packet scheduling can help to avoid the loss problem. When a streaming protocol trans-

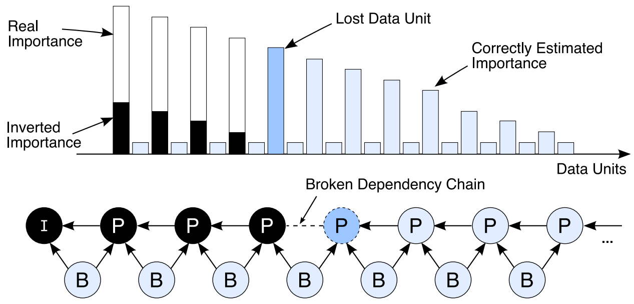


Fig. 4.14 : *Effects of packet loss on dependency chains: The loss of a data unit in the middle of a dependency chain partitions the chain and can lead to importance inversion.*

mits and repairs data units in importance order, then more important units are sent and repaired with larger probability than less important ones. The dependency model's object graph at the receiver side is less severely corrupted because destruction is likely to occur at the end-vertexes of dependency chains only. Hence, the chains become shorter, but inversion is prevented with a higher probability than it would be achievable with best-effort protocols.

Redundancy is a well known loss resilience tool for communication over erroneous channels. For group-based streams the dependency model already contains a simple form of redundancy in the data unit labels because type, group membership and dependency is repeated in every unit. Hence, it is sufficient to receive at least a single unit per group to reconstruct the dependency information for the total group. When further redundancy is required, the labels can be easily detached from data units and separately protected or transported. This is, however, beyond the scope of this thesis.

4.9 Experimental Evaluation

In order to evaluate the performance and accuracy of our dependency model we implemented a prototype in C++ and observed its behaviour with a large set of different H.264 video sequences under various simulated loss and visibility scenarios. For statistically sound and comparable observations we use standard video sequences, but due to their short duration and small resolution we also investigate longer video sequences with larger resolutions.

The sequences were encoded at variable bitrates (VBR), with several dependency patterns, epoch sizes and error-resilience features. Although some of the patterns are predictable, we do not examine the performance of the proposed prediction modes because they yield perfect estimation results anyway. We rather concentrate on worst case scenarios to analyse general limitations of dependency tracking systems. Furthermore, we do not consider constant bitrate (CBR) sequences because we do not expect them to raise further insight on dependency structures beyond VBR sequences. Although dependency relations in CBR settings can be adaptive and unpredictable, the number and patterns of data units is fixed. Hence, the estimation accuracy of our model under loss and limited visibility would degrade more sharply when compared to VBR sequences. Due to the lack of appropriate encoding software we could also not investigate layered and scalable streams. We assume that due to the complex dependency structures of scalable streams the estimations of our model would become more accurate, but we consider these experiments future work.

The first set of experiments focuses on comparing the accuracy of our importance estimation to MSE-based distortion metrics. We are interested in structural properties of bitstreams that influence our estimation accuracy such as, for example, the dependency pattern, the GOP size, and H.264 error resilience features. By linear regression analysis we show that our model performs well for a number of bitstreams, but we also reveal some deficiencies that originate in our simple assumptions about error propagation. A second set of experiments explores the conceptual limits of our dependency model, namely the effects of limited horizon size as well as the impact of loss on importance inversions. Finally, we provide performance figures for the Dependency Validation Service to show that dependency-based importance estimation is feasible at low complexity and in real-time settings.

4.9.1 Properties of Selected Video Sequences

Before reporting on results, we first introduce and justify the selection of video sequences used in our experiments. We also discuss the different encoder configurations, used to generate the H.264 bitstreams, and their impact on our dependency model.

Selected Sequences We used two sets of video sequences for the experiments. The first set contains standard sequences common in the video-coding and packet-video transport communities (table 4.3). These sequences can, for example, be obtained from the Video Traces Research Group from Arizona State University[2]. We call this set *standard*. Due to the short duration and limited resolution (QCIF and CIF) of the standard sequences we decided to use a second set with longer durations and larger resolutions up to HDTV (table 4.4). The second set was obtained from the BBC Motion Gallery[3], hence we call it *BBC*.

When available, we examined a sequence in multiple resolutions to detect effects that solely depend on the number of pixels or macroblocks per frame, while avoiding the considerable impact of content-specific features, such as low or high details, low or high local motion, and low or high global motion. Although content diversity is the most unpredictable and the least controllable property of a video sequence, it is a good indicator to bias the impact of controllable effects. Hence, when presenting results, we always show multiple sequences for comparison.

Both sequence sets have the following properties: In general, the standard set contains low resolution sequences without scene-cuts only, while the BBC set contains only high resolution sequences with scene cuts every 2 to 5 seconds. Most of the BBC sequences feature a considerable amount of motion and the HD sequences also a high amount of detail. Some sequences are noisy with respect to fluctuating pixel differences between consecutive frames. Noise means that encoding the source video is either less efficient or the quality of details suffers. Scene cuts and a high amount of motion (either local or global) also influence coding efficiency because more bits are required to encode the residual error after motion compensation. With respect to error propagation and the importance distribution, we expect our model to closely match the real distortion

[2]`http://trace.eas.asu.edu/`

[3]`http://www.bbcmotiongallery.com/Customer/Showreels.aspx`

	Resolution in		**Frames**	**Length**	**Content**
Name	**pixel**	**MB**	**& Rate**	**in sec**	**Characteristics**
Akiyo	(Q)CIF	11x9 / 22x18	300@25	12	very few motion
Coastguard	(Q)CIF	11x9 / 22x18	300@25	12	slow global motion
Foreman	(Q)CIF	11x9 / 22x18	300@25	12	local/global motion
Highway	(Q)CIF	11x9 / 22x18	2000@25	80	slow global motion

Tab. 4.3 : *The set of standard video sequences used in our experiments.*

Name	Resolution in pixel	Resolution in MB	Frames & Rate	Length in sec	Content Characteristics
CBS News	720x576	45x36	5185@25	207.40	scene cuts, very noisy
NHK	720x576	45x36	6295@25	251.80	cross-fadings and cuts, mostly slow global motion
Rip Curl	720x576	45x36	4621@25	184.84	fast global motion, many scene cuts
Science	720x576	45x36	2577@25	103.08	scene cuts, much and fast global and local motion
HD World	1266x720	80x45	2525@25	101.00	scene cuts, much global motion, noisy source
Wild Africa	1266x720	80x45	6744@25	269.76	scene cuts, slow mixed global and local motion, noisy source

Tab. 4.4 : *The set of BBC video sequences used in our experiments.*

because most of the frames are predicted and rely on other frames along the dependency chains. For high-motion sequences many macroblocks may be intra-coded rather than predicted even if the frame-type suggests prediction. This intra-updates may quickly attenuate error propagation and hence, dependency relations along a chain become weaker. Because our model currently ignores this fact for complexity reasons, it may perform worse if intra-updates are frequent.

Encoder Settings For generating H.264 bitstreams we used the H.264 reference encoder (JM12.2[4]) and the open-source x264 library[5]. In contrast to JM12.2, x264 provides no error resilience tools and it is only capable of producing a single slice per frame. Whenever error resilience was of no principal interest, we used x264 as encoder. JM12.2 was used to generate all error resilient streams, using forced intra-refresh for one line of macroblocks per frame. We also restricted the size of slices to the typical network-layer MTU of 1452 byte (Ethernet frame size minus IP+UDP+RTP headers). In the following we refer to the fixed slice mode as used by x264 as *slice mode A*, and to the network-adaptive slice mode as *slice mode B*. All streams were encoded with one IDR-picture per GOP to hard-limit error propagation. For all experiments we used variable bitrate streams to achieve the best possible quality at a given target bitrate and to generate variability in bitstream structures.

[4]`http://iphome.hhi.de/suehring/tml/download/`
[5]`http://www.videolan.org/developers/x264.html`

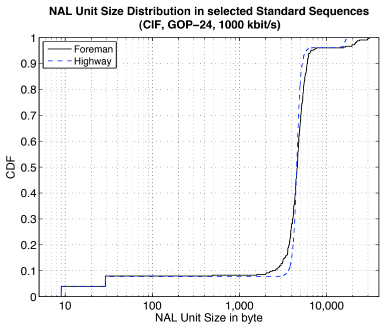
For the dependency structure we used fixed (I-P-B-P and fixed pyramid) as well as adaptive (I-P-P-P and adaptive pyramid) patterns. In pyramid patterns, some B-frames are used as references to form a dyadic tree structure of multiple levels. Because B-frames at each level are equally spaced, this format allows for smooth temporal scaling by removing all frames from one or more levels. In adaptive mode the encoder selects reference frames based on the content. We limited the number of references per frame to 8 and the radius for reference picture selection to 16. Streams with a fixed dependency structure are strictly predictable, however, the more error resilient network-adaptive slice mode B makes those streams unpredictable because it generates a variable number of NAL units per frame. For the standard video sequence set we selected

- two different dependency structures (adaptive I-P-P-P and fixed I-P-B-P),
- with and without error resilience (both slice modes),
- at GOP-sizes 12, 24, and 48 frames, and
- 10 target bit-rates between 100 kbit/s and 1000 kbit/s.

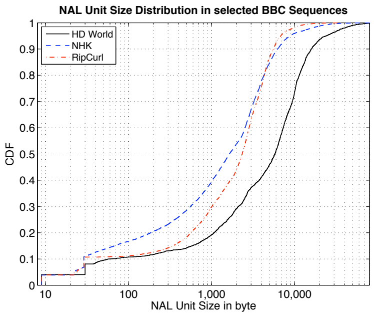
The BBC set of video sequences we encoded

- in four different dependency patterns (fixed I-P-B-P, fixed pyramid, adaptive I-P-P-P, and adaptive pyramid)
- with and without error resilience (for pyramid streams only without error resilience)
- at a common GOP-size of 24 frames and
- at 600 kbit/s bitrate for 720x576 resolution sequences and 3000 kbit/s for 1266x720 resolution sequences.

This gives a total number of sequences of $2 \times 2 \times 3 \times 10 \times 8 = 960$ for the standard set and a total number of sequences of $4 \times 2 \times 6 = 48$ for the BBC set. We consider these numbers sufficient for a sound statistical analysis.

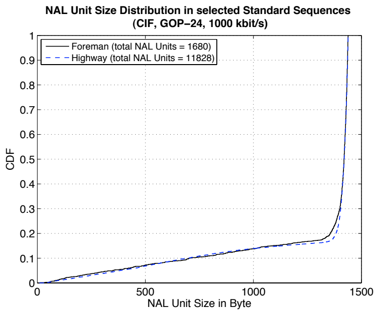


(a)

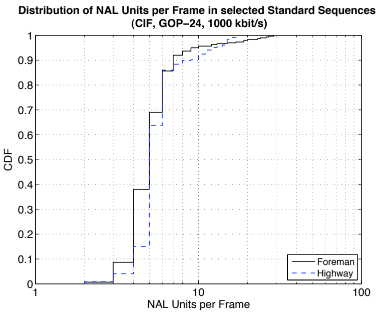


(b)

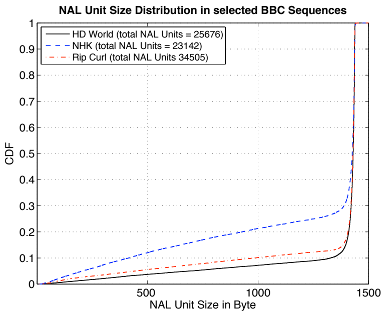
Fig. 4.15 : *Distributions of NAL Unit size in sequences with slice mode A.*



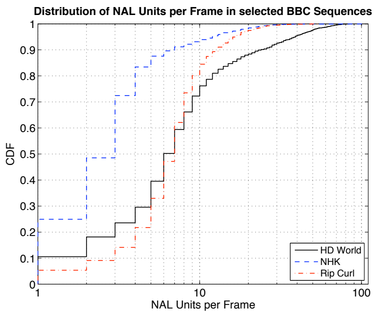
(a)



(b)



(c)



(d)

Fig. 4.16 : *Distributions of NAL Unit size for sequences in slice mode B.*

	Adaptive I-P-P-P Dependency		
Name	**GOP-12**	**GOP-24**	**GOP-48**
Akiyo[1]	5.61/26/4.99	5.61/ 31/ 3.70	5.61/21/1.88
Coastguard[1]	5.56/19/3.21	5.58/ 20/ 2.74	5.55/20/2.08
Foreman[1]	5.57/26/4.00	5.56/ 29/ 3.47	5.56/29/2.50
Highway[1]	5.90/15/2.53	5.91/ 18/ 2.62	5.90/18/2.06
CBSNews[2]	-	3.72/129/ 6.28	-
NHK[2]	-	3.68/105/ 4.80	-
RipCurl[2]	-	7.47/ 54/ 4.84	-
Science[2]	-	5.11/111/ 6.95	-
HDWorld[3]	-	10.16/ 84/12.17	-
WildAfrica[3]	-	10.10/177/15.23	-

Tab. 4.5 : *Number of NAL Units per video frame as (mean/max/std), encoded at* $1000kbit/s\,)^1$, $600kbit/s\,)^2$, *and* $3000kbit/s\,)^3$ *target bitrate. The resolution is CIF for standard sequences and resolution given in table 4.4 for BBC sequences.*

Stream Properties To further understand the properties of the encoded sequences we closely examined the bitstreams. We are mainly interested in the number and the distributions of data units (H.264 NAL units) per frame and per GOP because group semantics and horizon control in our model are sensitive to these properties.

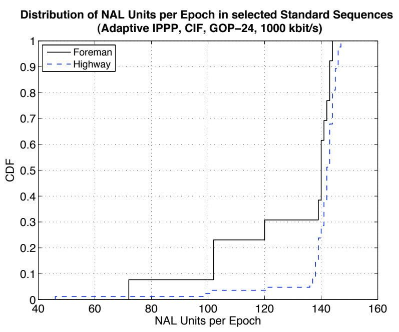
Figure 4.15 displays the distributions of NAL unit sizes for selected x264-encoded sequences from both sequence sets. Although, from a dependency perspective, the actual size of a data unit is irrelevant for importance estimation we provide the graphs for informational purposes. Please recall, that x264 supports a single slice per frame only (slice mode A). Hence, NAL units can become large, compared to the permitted network-level MTU (around 1500 bytes in the Internet [167]) or the maximum packet size of UDP (65,527 bytes [53]). We did, however, not consider fragmentation of large data units. Figure 4.16 shows the NAL unit size distributions for JM12.2 (slice mode B) to the left and the distribution of NAL units per frame to the right. The graphs show that a considerable amount of frames (more than 90 percent) is encoded in two or more NAL units and that 5 percent (standard set) to 30 percent (BBC set) of all frames contain more than 10 NAL units. It becomes also apparent that an increase in resolution translates into an increased variance of data units per frame. For comparison we also provide the exact figures of mean, max and standard deviation in table 4.5.

Name	Fixed I-P-B-P Dependency GOP-12	GOP-24	GOP-48
Akiyo[1]	4.12/24/5.11	4.12/ 29/4.28	4.07/20/2.92
Coastguard[1]	4.08/17/3.91	4.02/ 19/3.68	4.01/19/3.29
Foreman[1]	4.12/25/4.35	4.14/ 27/3.88	4.10/28/3.19
Highway[1]	4.07/15/3.20	4.06/ 18/2.99	4.05/18/2.46
CBSNews[2]	-	2.63/103/3.80	-
NHK[2]	-	2.62/ 59/3.57	-
RipCurl[2]	-	2.59/ 31/2.17	-
Science[2]	-	2.66/ 65/3.98	-
HDWorld[3]	-	6.57/ 80/8.11	-
WildAfrica[3]	-	6.82/125/9.77	-

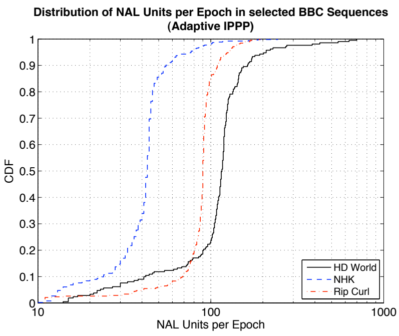
Tab. 4.6 : *Number of NAL Units per video frame (continued).*

Pattern	Name	GOP-12	GOP-24	GOP-48
Adaptive I-P-P-P	Akiyo[1]	67.76/74/8.71	130.23/ 147/ 23.06	241.86/294/75.35
	Coastguard[1]	67.16/75/8.60	129.54/ 147/ 22.85	239.29/287/73.44
	Foreman[1]	67.32/75/8.66	129.23/ 145/ 21.89	239.86/286/73.69
	Highway[1]	70.73/75/4.32	140.81/ 148/ 12.64	281.24/295/20.31
	CBSNews[2]	-	44.68/ 381/ 42.96	-
	NHK[2]	-	44.08/ 245/ 20.38	-
	RipCurl[2]	-	89.62/ 193/ 23.18	-
	Science[2]	-	61.31/ 431/ 49.05	-
	HDWorld[3]	-	121.69/ 698/ 79.44	-
	WildAfrica[3]	-	121.24/1367/129.35	-
Fixed I-P-B-P	Akiyo[1]	49.64/86/11.88	95.38/ 103/ 12.68	175.00/199/52.68
	Coastguard[1]	49.08/51/ 0.84	93.15/ 102/ 12.70	172.57/196/50.49
	Foreman[1]	49.60/54/ 1.91	95.85/ 102/ 9.79	176.43/198/46.79
	Highway[1]	48.78/52/ 1.80	96.83/ 101/ 6.74	193.19/199/10.08
	CBSNews[2]	-	126.45/ 596/ 62.54	-
	NHK[2]	-	125.21/ 515/ 46.71	-
	RipCurl[2]	-	123.56/ 513/ 46.64	-
	Science[2]	-	126.94/ 595/ 82.24	-
	HDWorld[3]	-	312.86/ 818/101.02	-
	WildAfrica[3]	-	326.44/1780/176.83	-

Tab. 4.7 : *Number of NAL Units per epoch as (mean/max/std), encoded at* $1000kbit/s\,)^1$, $600kbit/s\,)^2$, *and* $3000kbit/s\,)^3$ *target bitrate. The resolution is CIF for standard sequences and resolution given in table 4.4 for BBC sequences.*



(a)



(b)

Fig. 4.17 : *Distributions of epoch sizes for error-resilient sequences (slice mode B).*

The variable number of NAL units per frame in slice mode B directly translates into a variable number of NAL units per epoch as depicted in figure 4.17 and table 4.7. Please note that the x-scale for BBC set distributions in figure 4.17(b) is logarithmic. We later use the mean and the maximum values of the distributions to show how different horizon sizes below the mean and above the maximum influence the prediction accuracy of our model.

Although the standard sequences contain too few epochs for a sound statistical analysis (for 24 frames per GOP there are only 13 epochs in Foreman and 84 epochs in Highway), the figures in table 4.7 indicate similarity between mean and maximum values for all sequences across multiple GOP-sizes.

4.9.2 Accuracy of Dependency-based Importance

To figure out how close our estimated dependency-based importance matches traditional distortion metrics we performed a statistical analysis over different streams and dependency structures. Because several factors can influence the accuracy of the predicted importance values we systematically examine the effects of different dependency patterns, the effects of error-resilience tools, and the effects of different GOP-sizes, resolutions and encoding bitrates.

We compare our performance (a) to the popular Mean Squared Error (MSE) distortion metrics which is based on a per-frame pixel differences between a distorted sequence and a reference sequence and (b) to the Structural Similarity (SSIM) metrics [168] that takes human perception of visible distortion into account.

Methodology For calculating the actual MSE and SSIM distortion values for each NAL unit in a sequence we used the analysis-by-synthesis method proposed by Masala and deMartin [162]. According to this method we systematically removed one NAL unit and calculated MSE and SSIM metrics between the decoded sequence and a sequence that was not subject to loss. Note, that this method differs from encoder benchmarks where the original sequence (not the encoded one) is used as reference. This is because we are interested in the distortion that is introduced by network packet loss rather than the distortion introduced by lossy encoding.

For decoding the corrupted sequence we enabled the frame-copy and motion-copy error concealment modes provided by both x264 and JM12.2 decoders. The x264 sequences were actually decoded using MPlayer[6] because x264 provides no own decoder. When a frame was not concealed and output by the respective decoder, we copy-concealed its loss using the previous frame during distortion calculation. Due to the complexity of distortion computation and the length of our sequences we compared only the affected epochs up to the next intra-coded frame that restricts error propagation. While the MSE value was averaged over all affected frames, we use only the smallest SSIM value of any affected frame. Because SSIM values range from 0 (worst) to 1 (best) the smallest value characterises the frame with the largest distortion introduced by the simulated loss. Note that this may be the frame that misses a NAL unit or another frame that depends on the corrupted frame. The distortion calculations took a considerable amount of time on current high-end processors (Intel Xeon 3.2 GHz), emphasising the practical limits of traditional distortion metrics when time or resources are scarce such as in live streaming scenarios and on mobile devices.

The dependency-based importance values were calculated by our DVS prototype implementation. For initialisation we used the H.264 dependency description in listing A.2 and the maximum epoch sizes in table 4.7. The horizon size was configured to cover at least two complete epochs of maximal size to allow each data unit to reach steady-state. Because our model requires the actual dependency between data units,

[6] `http://www.mplayerhq.hu`

but provides no means to extract these properties from a bitstream, we also developed a tool to obtain the desired properties from H.264 sequences and store them in trace files.

Performance Metrics To identify a connection between importance and distortion metrics we used the following correlation analysis tools:

Spearman Rank-Order Correlation Coefficient (SROCC) is an analytical tool that reveals if an increase in dependency-based importance is justified by an increase in distortion and thus if predictions made by the dependency model are relevant at all. Although rank-order is just a qualitative statement rather than a quantitative one, it would be already sufficient for priority-based packet scheduling and adaptive error control because decisions in such schemes are based on ordering relations between data units.

Pearson Product-Moment Correlation Coefficient (CC) is an analytical tool that reveals in addition to the qualitative statement of rank-order if an increase in distortion is proportional to an increase in importance. If so, the dependency model can also predict the absolute importance of each data unit over the complete sequence. This might be an important property when absolute quality levels are of interest, such as in performance-critical applications that must preserve a pre-determined quality and fail otherwise.

Scatter-Plots are a graphical tool that reveals relationships or association between importance and distortion values. We plot the dependency-based importance at the horizontal axis and distortion as the response variable on the vertical axis. Note that the scatter plots we use always contain normalised data. Comparing their absolute values is therefore meaningless.

Quantile-Quantile-Plots (*Q-Q plots*) are a graphical tool that provides more insight into the nature of the distributions of importance and distortion values. QQ-Plots show if and where both distributions are similar and where they are not. For comparison we plot the importance value distribution on the horizontal axis and the distortion value distribution on the vertical axis. The reference line in a QQ-plot reveals if the dependency model underestimates or overestimates real distortion values. As with the scatter plots, comparing absolute values is meaningless.

For sound statistical analysis we removed extreme outliers and illegal values that resulted from decoder crashes and normalised the data sets using the Box-Cox transformation [169].

Influence of the Dependency Pattern We start our comparison with observations of different dependency patterns because this is the most prominent property of media streams and because it suggests there is an obvious relationship to dependency-based importance.

To isolate all other effects we use only streams in slice mode A, encoded with the same GOP-size (24) and at a single bitrate (1000 kbit/s for standard sequences, and 600 resp. 3000 kbit/s for BBC sequences). We vary two parameters only, namely the dependency pattern and the resolution. We selected two fixed patterns (Fixed I-P-B-P and Fixed Pyramid) where the dependency was pre-determined by the encoder configuration and two adaptive patterns (Adaptive I-P-P-P and Adaptive Pyramid) where the encoder selected reference pictures based on content-specific properties and bitrate constraints.

As expected, the correlation results in table 4.8 and table 4.9 indicate a strong relationship between importance and MSE distortion, especially for fixed dependency patterns. Both, the rank-order and the Pearson correlation coefficients show an equal tendency. The quantitative relationship expressed by the Pearson correlation coefficient is less evident when the MSE distortion data set is not normalised. The rank-order correlation is not affected by these effects.

In general, dependency-based importance values are natural numbers that reflect the length of dependency chains, while distortion metrics are real numbers in a much finer resolution. Moreover, exact pixel-level distortion depends on more effects than dependency and error propagation alone. Hence, it is plausible that importance cannot completely explain distortion effects, but we conclude that our dependency model predicts at least the order of importance values sufficiently accurate.

We also find from tables 4.8 and 4.9 that correlation is strong for MSE-based distortion metrics, but worse for the SSIM metrics although SSIM explains the perceived distortion more accurate than the simple MSE. While for small resolutions there is a clear relation, for large resolution the fit ceases to exist. We suspect our method of applying SSIM to video sequences is wrong because we did not adjust it to the image resolution as suggested. In fact, the SSIM method we used was originally developed for still images. It does not consider motion and sequences of multiple

		MSE			SSIM		
Pattern	**Stream**	SROCC	CC	OR	SROCC	CC	OR
Fixed I-P-B-P	Akiyo (QCIF)	0.821	0.735	0.092	-0.570	-0.566	0.092
	Coastguard (QCIF)	0.875	0.829	0.092	-0.709	-0.644	0.092
	Foreman (QCIF)	0.795	0.740	0.098	-0.410	-0.397	0.098
	Highway (QCIF)	0.829	0.765	0.101	-0.632	-0.574	0.101
	Akiyo (CIF)	0.822	0.736	0.092	-0.582	-0.556	0.092
	Coastguard (CIF)	0.891	0.846	0.092	-0.719	-0.650	0.092
	Foreman (CIF)	0.844	0.783	0.092	-0.422	-0.400	0.092
	Highway (CIF)	0.838	0.774	0.101	-0.523	-0.469	0.101
Adaptive I-P-P-P	Akiyo (QCIF)	0.540	0.572	0.089	-0.049	-0.113	0.089
	Coastguard (QCIF)	0.843	0.817	0.089	-0.200	-0.174	0.089
	Foreman (QCIF)	0.633	0.622	0.089	-0.187	-0.175	0.089
	Highway (QCIF)	0.708	0.662	0.107	-0.266	-0.270	0.107
	Akiyo (CIF)	0.598	0.617	0.089	-0.104	-0.119	0.089
	Coastguard (CIF)	0.891	0.848	0.089	-0.056	-0.056	0.089
	Foreman (CIF)	0.718	0.705	0.089	-0.143	-0.152	0.089
	Highway (CIF)	0.724	0.681	0.101	-0.092	-0.089	0.101

Tab. 4.8 : *Performance comparison between dependency-based importance and other estimation models for standard sequences. The sequences are encoded with x264 at GOP-size 24, 1000 kbit/s and in slice mode A. SROCC: Spearman Rank Order Correlation Coefficient; CC: Pearson Product-Moment Correlation Coefficient; OR: Outlier Ratio; Note that because SSIM ranges from 1 for the best quality to 0 for the worst quality, the correlation coefficients are expected to be negative when there is a linear relationship to importance.*

pictures. Hence, in our simulations we only used the worst quality image as candidate to compute SSIM metrics and used this value as a distortion measure. While there exists a video extension to SSIM [170] we sacrificed its implementation in favour of a broader analysis using the less complex MSE metrics. A thorough comparison to a suitable SSIM metric remains an open issue.

Our dependency model seems to perform better for fixed dependency patterns than for adaptively selected reference (4.18(a)), especially for patterns with more B-frames (the pyramid structures, figure 4.19(b)). A reasonable explanation for less accurate estimations of adaptive dependency patterns is that although the dependency model exactly tracks existing references the actual prediction mechanisms used in H.264 perform a weighted reference selection. The dependency model does not account for weighted references, although it would be possible to attach weights to vertices and edges in the dependency graph.

		MSE			SSIM		
Pattern	**Name**	**SROCC**	**CC**	**OR**	**SROCC**	**CC**	**OR**
Fixed I-P-B-P	CBS News	0.651	0.603	0.083	-0.364	-0.390	0.083
	NHK	0.580	0.542	0.082	-0.282	-0.297	0.082
	Rip Curl	0.735	0.648	0.103	-0.300	-0.303	0.103
	Science	0.527	0.479	0.116	-0.298	-0.312	0.116
	HD World	0.634	0.563	0.141	-0.284	0.028	0.141
	Wild Africa	0.594	0.548	0.104	-0.103	0.001	0.104
Fixed Pyramid	CBS News	0.575	0.581	0.108	-0.434	-0.463	0.108
	NHK	0.532	0.545	0.087	-0.226	-0.002	0.087
	Rip Curl	0.595	0.584	0.092	-0.426	-0.419	0.092
	Science	0.508	0.511	0.110	-0.407	-0.445	0.110
	HD World	0.614	0.609	0.082	-0.445	-0.453	0.082
	Wild Africa	0.583	0.575	0.096	-0.340	-0.350	0.096
Adaptive I-P-P-P	CBS News	0.668	0.606	0.083	-0.390	-0.401	0.083
	NHK	0.551	0.495	0.080	-0.265	-0.269	0.080
	Rip Curl	0.713	0.604	0.099	-0.286	-0.285	0.099
	Science	0.576	0.514	0.115	-0.379	-0.370	0.115
	HD World	0.591	0.511	0.150	-0.264	-0.251	0.150
	Wild Africa	0.547	0.467	0.098	-0.205	-0.185	0.098
Adaptive Pyramid	CBS News	0.747	0.669	0.129	-0.523	-0.491	0.129
	NHK	0.660	0.581	0.166	-0.272	0.005	0.166
	Rip Curl	0.689	0.594	0.105	0.000	0.000	0.105
	Science	0.652	0.564	0.113	-0.484	-0.421	0.113
	HD World	0.656	0.568	0.084	-0.346	-0.298	0.084
	Wild Africa	0.606	0.532	0.092	-0.276	-0.235	0.092

Tab. 4.9 : *Performance Comparison between Importance Estimation Models with different Dependency Patterns for BBC Sequences. SROCC: Spearman Rank Order Correlation Coefficient; CC: Pearson Product-Moment Correlation Coefficient; OR: Outlier Ratio*

The differences between adaptive and fixed patterns seem stronger for low-resolution standard sequences (see figure 4.18) while they decrease for higher resolution BBC sequences (displayed in figure 4.19). This effect is, however, not justified by a change in resolution. Figures 4.18(a) and 4.18(b) reveal that there are no significant differences between QCIF and CIF resolution at least for standard sequences. This leads us to the conclusion that the observed effects of lower correlation depend on content-specific properties such as motion and detail. Especially for the BBC set with high motion and many scene cuts our importance estimations become worse.

The results disprove our initial expectation that low motion sequences are better estimated because more references and less intra-coding of differences between consecutive frames is used. In fact the Akiyo sequence

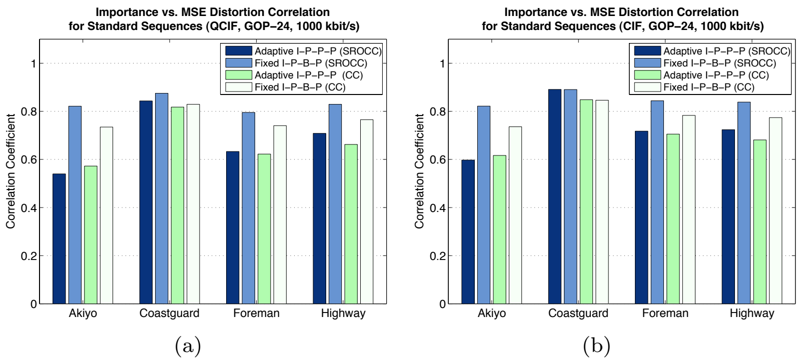


Fig. 4.18 : *Correlation between dependency-based importance and MSE distortion for different dependency patterns and resolutions of standard sequences. GOP-size and bitrate are constant.*

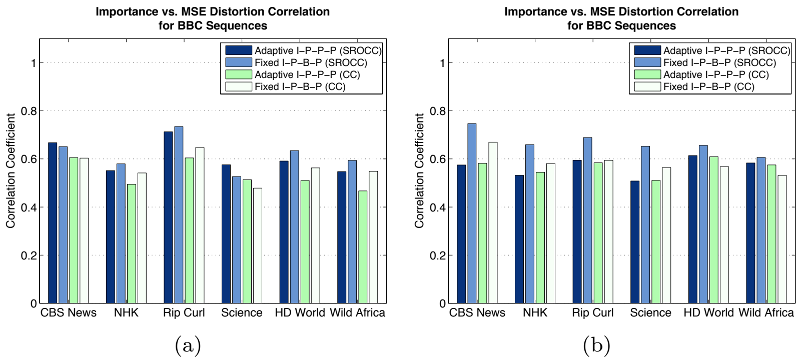
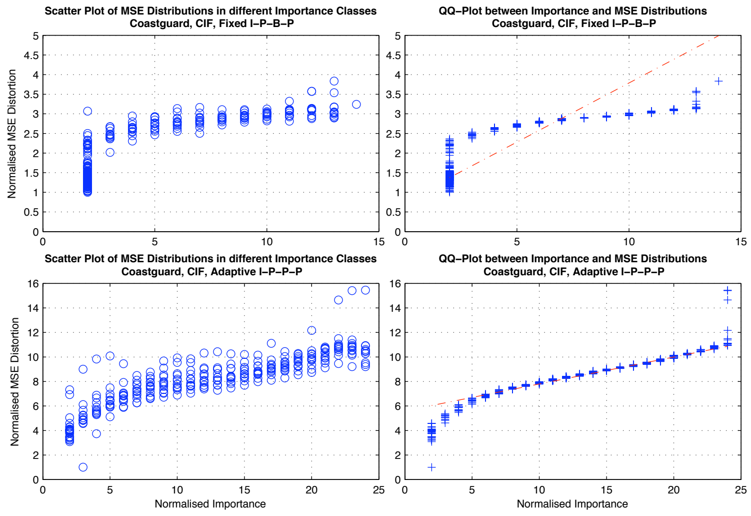
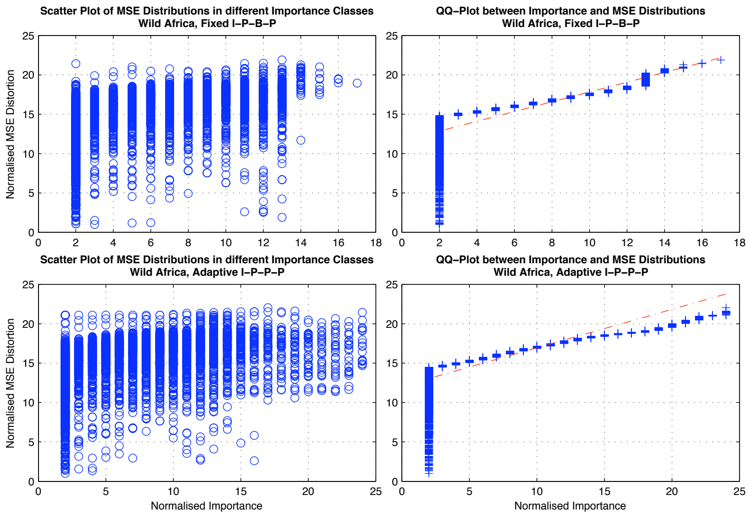


Fig. 4.19 : *Correlation between dependency-based importance and MSE distortion for different dependency patterns of BBC sequences. GOP-size and bitrate are constant.*

in figure 4.18 reveals that our model performs worse for adaptive coded low motion sequences. We explain this unexpected effect with the error concealment method employed at the decoder. Because of the few differences between frames, frame-copy error concealment performs outstandingly well here. We intentionally avoided modelling of concealment schemes because the actual method employed at the decoder may be unknown to the sender and application-level concealment should remain out of scope of a transport-level dependency-tracking framework anyway.



(a)



(b)

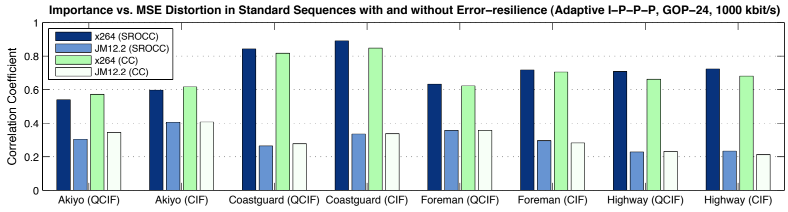
Fig. 4.20 : *Detailed Scatter- and Q-Q-Plots for Coastguard (CIF, GOP-24, 1000 kbit/s) and Wild Africa (1266x720, GOP-24, 3000 kbit/s), both encoded in slice mode A without error-resilience.*

Apparently, the higher resolution is no sufficient explanation for decreased correlation, because the BBC sequences approach correlation values also found at QCIF resolutions. Hence we conclude that the different qualities of fit stem from content-specific attributes such as scene cuts, motion, detail and noise. To gain further insights we plot the normalised MSE distortion values as a response to the importance into a scatter plot and a Q-Q-Plot. If the assumed correlation exists, the points would arrange along a straight line or a curve. Figure 4.20(a) displays the diagrams for a sequence with a very good fit and figure 4.20(b) shows the plots for a worse fit. The scatter plots show a considerable overlap of distortion values between distinct importance classes. There is, however, a general tendency of higher distortion values towards the upper importance classes. For fixed dependency structures the variance of distortion values is smaller than it is for adaptive structures. For the Wild Africa sequence the variance is considerably larger than for the Coastguard sequence.

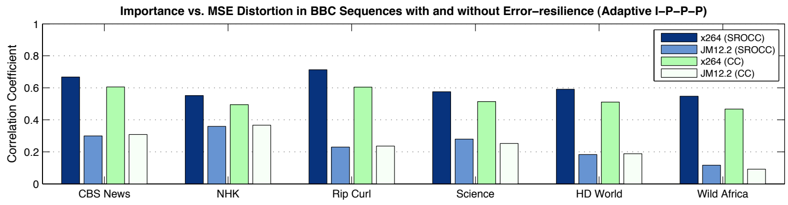
The shape of the curves also reveal a non-linear relationship between importance and distortion. Especially for very small and for very large values both metrics show different distribution characteristics, as the tails in the Q-Q-Plots indicate. The Q-Q-Plots further show that there are much more small values in the importance distribution than the MSE distribution contains. For example, the dependency model assigns the value 2 to data units who cover a large range of distortion values (30-50% of all values for I-P-B patterns). From the general structure of the dependency graph we know that the value 2 is only assigned to the least important units, such as B-frames and the last P-frames in an epoch. From a scheduling perspective, assigning such small values is reasonable because the affected units are not used as references. When, however, a scheduler must choose one out of many least important data units to send or repair, the dependency model provides no further advice.

We conclude, that importance estimation achieves a close match to MSE distortion in the centre of the distributions but at the tails, in particular the lower tail, importance values do not fit. Hence, we suggest that an future improvements should concentrate on the distribution tails to increase the fit.

Influence of Encoding-Layer Error-Resilience Tools The accuracy of dependency-based importance metrics may decrease when encoder-side error resilience tools are used because they influence the propagation of



(a)

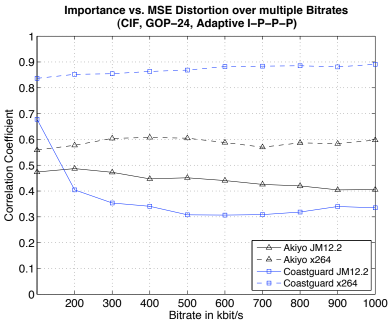


(b)

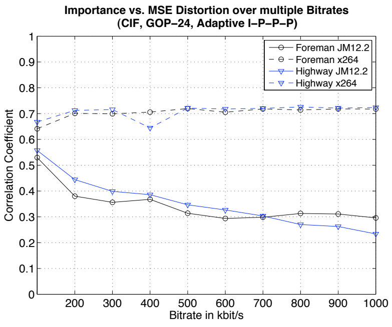
Fig. 4.21 : *Correlation between dependency-based importance and MSE distortion for different slice modes, with (JM12.2) and without (x264) error resilience. Dependency pattern, resolution, GOP-size and bitrate are constant.*

errors along dependency chains. Therefore we investigate the prediction quality of our model for error resilient sequences. We selected the network-adaptive slice mode B and forced intra-refresh of one line of macroblocks per frame. We also considered H.264 data partitioning, but decoder failures prevented us from calculating distortion values for all NAL units. Hence, we dropped the set of data partitioned streams from this evaluation.

Figures 4.21(a) and 4.21(b) reveal that the effects of error resilience have a large impact on the fit between importance and distortion metrics. We use the bars from figures 4.18 and 4.19 as reference. At high resolutions and for complex streams with a lot of motion and details, in particular in HDTV streams, the importance metric does not fit MSE distortion anymore. The distortion seems to become random, depending on the content and the size of the slice that is transported in a NAL unit. The same effect is visible for increased bitrates in figure 4.22. Although the ratio of intra-updates remains constant the number of bits per frame increases, generating in more NAL units that cover smaller spatial regions. This leads us to the conclusion that the number of encoded macroblocks per slice is the crucial factor for the fit.



(a)



(b)

Fig. 4.22 : *Correlation between dependency-based importance and MSE distortion for different bitrates. Dependency pattern, resolution and GOP-size are constant.*

This is plausible because only small regions of a frame are lost when a NAL unit is missing. The distortion after decoding and error concealment heavily depends on the lost content and on future references to pixels at the lost macroblock positions. Hence, prediction, motion compensation and concealment may or may not perform well at the decoder, resulting in content-specific (random) MSE values. The assumptions of the dependency model that dependency exists between frames only and that fragments (NAL units) always contribute to a total frame does not capture this fine-grained relationship at the macroblock level. The assumption is, for example, expressed in the `equal` group semantics we defined in the H.264 dependency description in listing A.2.

The scatter plot in figure 4.23 supports our conclusion. The variance of distortion values per importance class remains high over the total range although there is a tendency of low distortion values towards low importance classes and a tendency of the largest distortion values towards upper importance classes. While the centre of both distributions shows a linear relationship in the Q-Q plot, the heavy tails reveal that our dependency model over-estimates small distortion values and under-estimates large values. This effect is in particular visible for streams with large group sizes (numbers of NAL units per frame) because the dependency model assigns equal importance values to all group members.

The effects of intra-updates seem to have less impact here because an increase in resolution decreases the intra-refresh rate (it is fixed to one

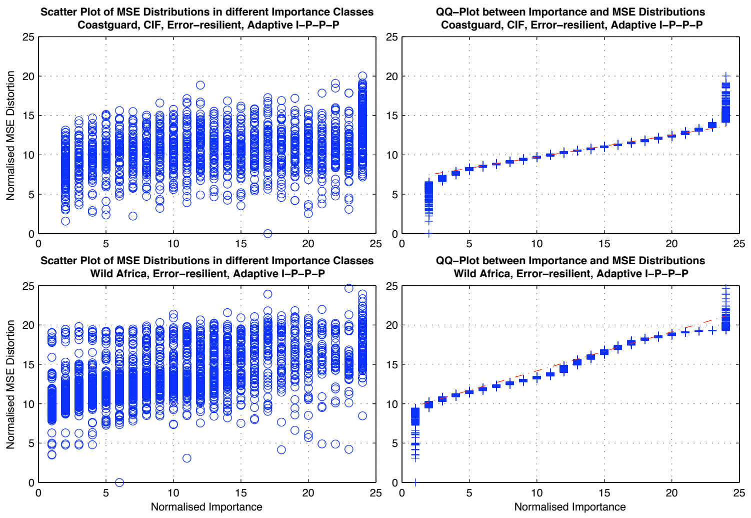


Fig. 4.23 : *Scatter- and Q-Q-Plots for error-resilient versions of Coastguard (GOP-24, 1000 kbit/s) and Wild Africa (GOP-24, 3000 kbit/s). The mean/max group size of Coastguard is 5.58/20 and 10.10/177 for Wild Africa.*

line of macroblocks in JM12.2). Although the number of macroblocks considerably increases at higher resolutions (see tables 4.3 and 4.4) the correlation steadily decreases.

A possible extension to improve the fit of our importance metrics would be to attach correct importance offset values to data units and to choose the `unequal` group semantics (see also section 4.8.2). This, however, requires knowledge about the expected distortion per data unit which contradicts the goals of the dependency model.

Influence of the GOP Size Our final observations regard the impact of different GOP sizes on the accuracy of importance estimations. We expect larger GOP sizes to create longer dependency chains and hence more detailed importance values. At least for streams in slice mode A the fit of our importance metrics should increase with the GOP size, while for error resilient streams we expect the effects of small NAL units to be stronger. We compare streams encoded with fixed and adaptive dependency pattern in slice mode A and slice mode B. All streams have equal bitrates (e.g. 1000 kbit/s for standard sequences) and a fixed resolution (e.g. CIF for standard sequences).

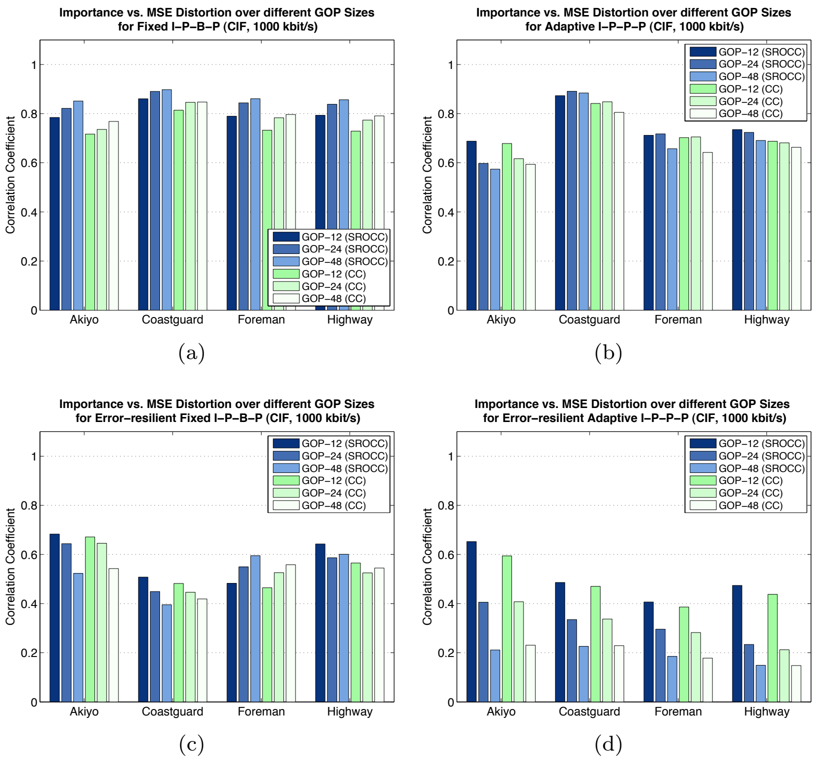


Fig. 4.24 : *Correlation between dependency-based importance and MSE distortion for different GOP-sizes. a,b) non- error-resilient slice mode A; c,d) error-resilient slice mode B; resolution and bitrate are constant.*

As expected, the correlation slightly increases for fixed dependency patterns in slice mode A (figure 4.24(a)), but it slightly decreases for adaptive patterns (figure 4.24(b)). The effects are, however, negligible. Interestingly, in error-resilient encoded streams with fixed dependency patterns (figure 4.24(c)) the influence of content-specific attributes seems to be stronger. Two streams, Foreman and Highway, show an inconsistent and unexpected behaviour we explain later.

In contrast, the weaker relationship between importance and distortion is more pronounced in error-resilient streams with adaptive reference selection (figure 4.24(d)). While the effects of multiple NAL units per

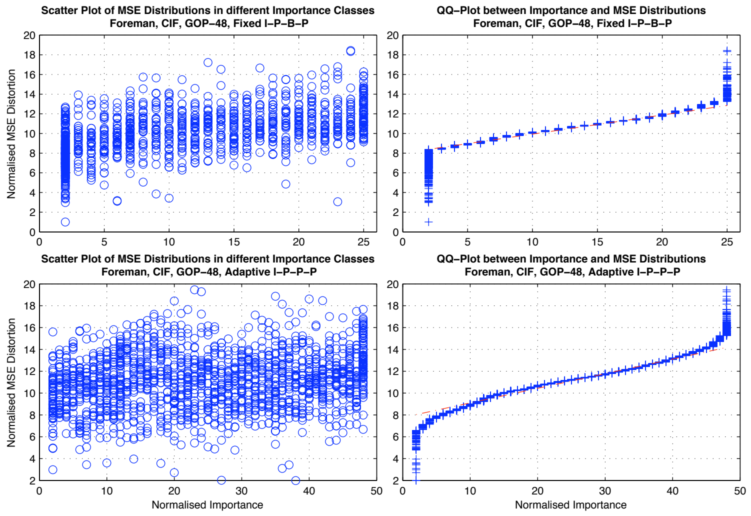


Fig. 4.25 : *Detailed Scatter-Plot and Q-Q-Plots of the error-resilient versions of Foreman (CIF, GOP-48, 1000 kbit/s), with fixed dependency and B-frames (upper figures) and with adaptive dependency and without B-frames (lower figures).*

frame remain constant, the effects of forced intra-refresh become visible. Over all GOP sizes the mean NAL unit number per frame is approximately stable as table 4.5 displays, but the larger GOP size allows the encoder to place more updates. At CIF resolution, every 18 frames one line of macroblocks is intra-refreshed. This gives at least one complete frame update for GOP size 24 and at least two complete frame updates for GOP size 48, resulting in much lower error propagation probability.

While intra-updates affect streams regardless of their dependency structure, structures that use B-frames are better predicted by our model than adaptive structures without B-frames. The reason is that intra-updates which are performed for a B-frame do not further limit error propagation because B-frames are not used as references. In our fixed dependency examples, B-frames represent half of all frames. Hence, half of the distortion value variability caused by intra-updates is attributable to B-frames alone. Since our model classifies them as least important the dependency/importance correlation for remaining frames increases. This effect is displayed in figure 4.25. While apart from their dependency pattern both sequences are equal, the variability over the range of importance classes differs considerably. This is because in the upper sequence all NAL units of each second frame are assigned to the lowest importance class (the stack of values to the left). For the lower sequence, these NAL units are spread across the total range of importance classes according to the frame position in each GOP.

The dependency model can only deduce intra-updates from data unit labels, but unfortunately they are not visible in slice headers. Even if a total slice is intra-updated, the slice type and its reference lists may not reflect this because the slice type in H.264 is not chosen according to the contained macroblocks. Instead, the allowed prediction modes per macroblocks are chosen according to the slice type. Although intra-update was enabled we found no difference when analysing NAL unit and slice headers of the encoded bitstreams. Hence, our H.264 parser did not translate intra-updates into empty reference lists in data unit labels. Therefore, the dependency model misses the intra-updates completely.

4.9.3 Effects of Limited Horizon Size

We already showed that the visibility of reference relations between data units can influence the importance estimation and can even lead do importance inversions. This is in particular critical for transport protocols and scheduling mechanisms that operate on a limited number of data units in a transmission window. Every wrong scheduling decision, based on false importance estimation decreases the overall quality of a transport mechanism. Hence, it is desirable to know whether and how it is possible to avoid negative effects.

Methodology In order to observe the magnitude of inversion effects we ran simulations with multiple horizon size limits and multiple unpredictable streams with regular and irregular dependency patterns. For sound statistics we only used sequences from the BBC set because they contain considerably more data units than the standard sequences.

For our simulations we used two DVS instances, one for reference purposes and the other for simulating a limited horizon. Both instances were initialised with the H.264 type description in listing A.2. The horizon of the reference DVS was configured to hold two maximal epoch sizes of data units more than the simulation DVS. The garbage collection strategy was set to `MaxHorizon` for both DVS instances. In order to construct a worst case scenario we disabled prediction and did not use importance offset values. The horizon size of the simulation DVS was varied between 25 percent and 400 percent of the mean epoch size of the measured stream (see table 4.7).

New data units were inserted into the reference DVS first and after a pre-roll delay of one maximum epoch size they were inserted into the simulation DVS too. This method ensured that for all data units in the simulation DVS the corresponding units in the reference DVS reached steady-state. In every simulation step, we first inserted one new data unit into both DVS instances (not the same unit due to the pre-roll delay) which led to the deletion of one old data unit by our garbage collector. Then, we obtained the importance values for all units in the simulation DVS and for the corresponding units in the reference DVS and stored them in two lists. We then ordered both lists by importance (and by sequence number when the importance was equal) and checked the resulting order for inversions. A unit u_i in the simulated list was considered inverted if any unit that succeeds u_i in the reference list preceded u_i in the simulated list. A scheduling algorithm that relied on a inverted order would perform a wrong decision.

Note that this scheme does not rely on absolute importance values, which are not comparable due to the different horizon sizes. Note also that we only counted inversions when a unit was degraded, not when a unit was more important than it should be. This strategy was chosen to attribute inversion to discriminated units only.

Metrics For each unit we counted all steps in which the unit's importance was inverted and divided this count by the horizon size to obtain an *inversion ratio*. The ratio tells which amount of time the importance of a unit was inverted compared to the time the unit was considered at all. Because inversions can only happen until a unit has reached steady-state the ratio describes a continuous fraction of time starting when the unit becomes first visible. The inversion ratio is 1 for units that were inverted during the whole interval and 0 for units that were never inverted.

Expectations We expect that importance inversions are more frequent for important data units because they much longer lack some of their transitive references than less important units (see also section 4.8.2). This happens more often to all leading data units in an epoch when data units from earlier epochs are still visible (e.g. the I-frame and the early P-frames). This is because all units from a previous epoch already reached steady-state while units in the current epoch have not.

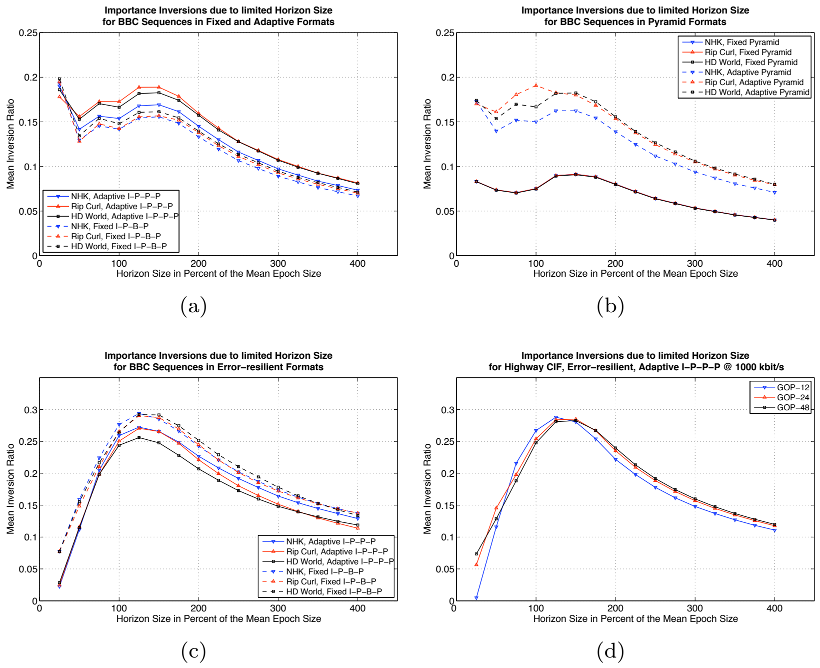


Fig. 4.26 : *Impact of horizon size on importance inversions. The ratio qualifies the time a data unit was inversed compared to the time it was visible.*

The maximum inversion ratio for any data unit is 0.5 because regardless of the horizon size an inversion can only occur across epochs. When the horizon size is less than the mean epoch size, most units leave the horizon before their importance reached steady-state. When, however, the horizon is larger than a single epoch, all units from at least one epoch are in steady-state. In any case, a data unit can be inverted at most until it reached the middle of the horizon. Either more important units from a previous epoch have left the horizon then or the unit has reached steady-state meanwhile.

Results Figure 4.26 shows how the horizon size influences inversions in streams with different dependency patterns and GOP sizes. We selected representative sequences and averaged the inversion ratio over all data units in a stream.

In general, the behaviour of inversions is as expected. When the horizon is too small to hold all data units from one epoch there are some inversions, but most data units leave the horizon before they are in steady-state. With an increase in horizon size the inversion ratio increases as expected because the more important data units are inverted longer. After a top between 100 and 150 percent of the mean epoch size the ratio drops again because the fraction of time where units are not inverted becomes larger now. At 200 percent, for example, even the most important I-frames are inverted only half of the time they are visible. When the non-inverted time becomes sufficiently large at very large horizon sizes, the ratio asymptotically approaches zero (not shown). The curves also reveal that regardless of the horizon size some data units experience at least a small period of inversion. The GOP-size seems to have no impact (see figure 4.26(d)).

In slice mode A, inversions for dependency patterns with B-frames are lower than for comparable patterns without them. B-frames are not used as references (in pyramid formats this applies only to the lowest layer of B-frames) and hence they are never inverted. The large number of unreferenced B-frames in our experimental patterns (0.5 for fixed I-P-B-P formats and 0.475 for pyramid formats) lessens the overall inversion ratio.

The different shapes of the curves between slice mode A (fig. 4.26(a) and 4.26(b)) and B (4.26(c)) at very small horizon sizes (less than one mean epoch size) stem from the fact that slice mode B uses variable sized groups and group members are considered equally important. While for slice mode A more units see inversions half of their time in the horizon, the probability for units in slice mode B to share the horizon with more important units is lower because more units are removed by garbage collection when groups are larger than the mean group size. This fact is also supported by figure 4.27 which shows the mean inversion ratio for data units in a particular importance class. At low and medium horizon sizes, sequences with multiple data units per frame experience less frequent inversions for low-priority data units.

At very small horizon sizes only the most important units are affected by inversions, while low important units experience no inversions. At horizon sizes larger than one mean epoch size, the inversion ratio linearly depends on the steady-state importance of a data unit. This is reasonable because more important units need to wait longer before all transitive references become visible. Different GOP-sizes do not significantly change this situation.

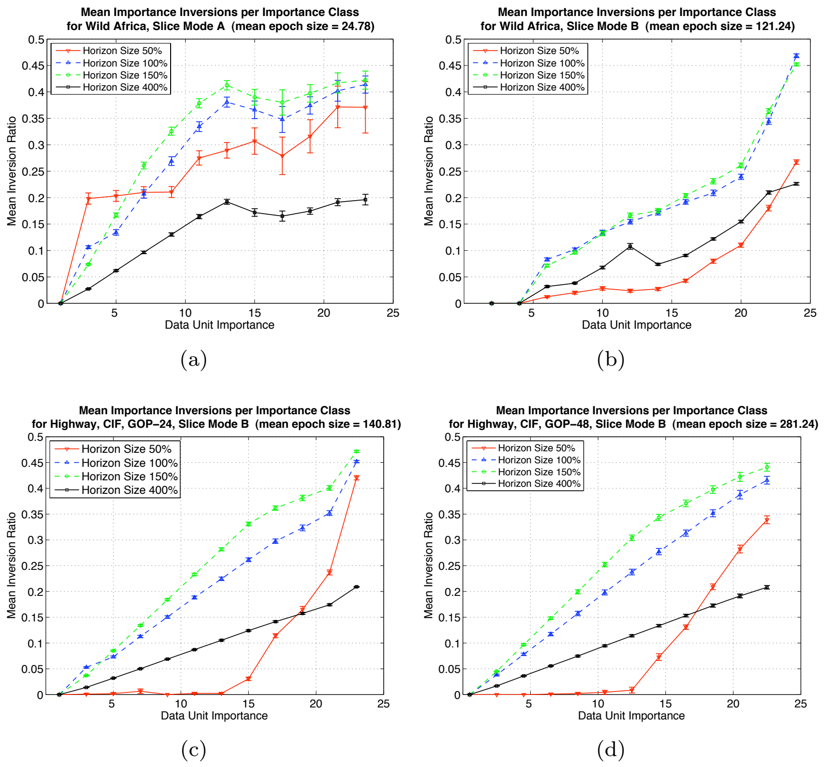


Fig. 4.27 : *Inversion Ratios per Importance class in different slice modes (a, b) for different GOP-Sizes (c,d). Sequences are encoded with adaptive I-P-P-P pattern at 1000 kbit/s.*

At the first glance, short horizons seem attractive because early deletion of units lowers the probability of inversions. Unfortunately, the most important data units are of particular interest to a scheduling scheme and these are the units that suffer most from invisibility, regardless of the horizon size. In order to hide inversions from scheduling mechanisms we recommend to use trusted importance values only. This means that a scheduler should wait at least until a data unit reached the middle of the horizon because then, the probability that the unit has reached steady-state is maximal. The minimum horizon size should be two times the size of the transmission or scheduling window of a streaming protocol.

4.9.4 Effects of Packet Loss

Unfortunately the horizon size is not the only source of importance inversions. Loss can also affect the visibility of dependency relations and lead to broken dependency chains. While we already provided general solutions to deal with loss in section 4.8.3, in this section we investigate the impact of different loss patterns on estimation accuracy. In particular, we are interested to examine how the steady-state importance of data units is affected by isolated and burst loss of data units.

Methodology The simulation setup differs from the horizon tests because now the steady state importance matters, rather than the history of inversions. Hence we set the horizon size to two times the maximal epoch size of a sequence to capture at least two consecutive epochs (see table 4.7). We choose this value in order to hide the effects of limited horizon size as shown in the previous experiment.

The loss pattern is modelled by a 2-state Gilbert Model [171] with a variable good-to-bad transition probability P, a bad-to-good transition probability Q, and a loss probability $p_{bad} = 0.9$ when in bad state. Q is chosen to generate loss bursts of average length 2, 5, and 10, and different values of P are used to simulate variable loss rates up to 20 percent.

We check every data unit for inversion when the data unit is in the middle of the horizon. At this point a data unit has reached steady-state because even for the largest epoch, all dependent units are visible. As for the horizon tests, a reference DVS is used to obtain steady-state importance of an uncorrupted stream as benchmark. For a sound statistical analysis we selected only sequences from the BBC set. We ran 20 iterations per experiment and calculated confidence intervals based on the Student-t distribution.

Metrics As performance metric we use the steady-state inversion rate of a total sequence, that is the number of data units that were inverted when they reached the middle of the horizon divided by the total number of data units in a particular sequence. In contrast to the per-unit inversion ratio, the steady-state inversion rate is a cumulative measure that has a meaning for the complete sequence only.

Expectations Generally, we expect a decrease in estimation accuracy with increased loss rate because more dependency relations are lost when

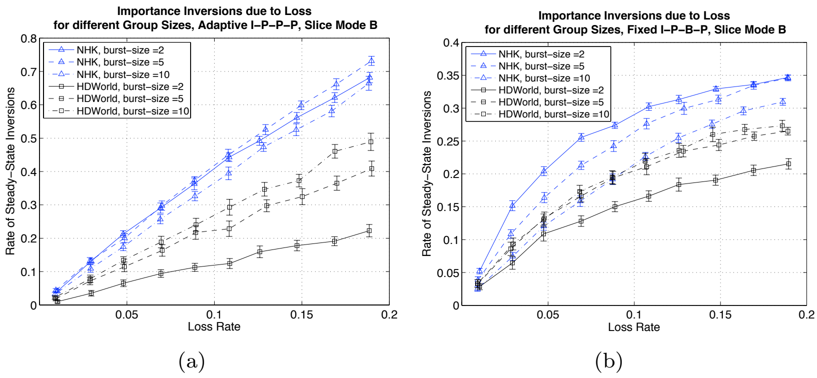
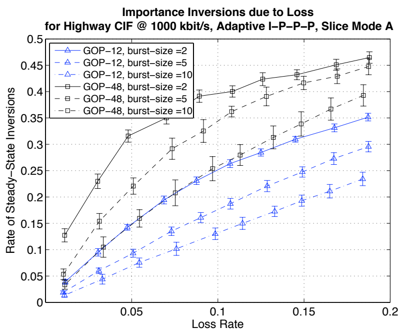


Fig. 4.28 : *Steady-state inversions over multiple loss rates for BBC sequences with different distributions of NAL units per frame.*

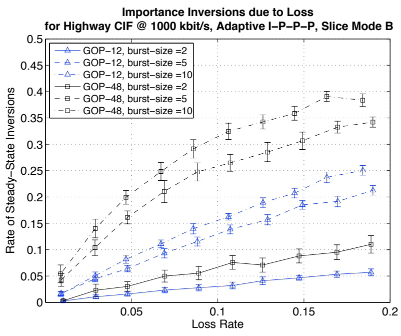
more data units are lost. Distinct stream formats have, however, a different loss resilience based on dependency pattern and redundancy, namely the number of data units per group. We expect sequences with long GOPs and with small groups to be more vulnerable to loss, especially large loss bursts.

Results Figure 4.28 shows that whether or not the dependency pattern uses B-frames, a sequence with many small groups experiences more steady-state importance inversions than a sequence with larger groups (see also figure 4.16(d) for the group size distributions). Around 50 percent of all frames in NHK contain two or less NAL units and only 7 percent of all frames are larger than 10 NAL units, while only 18 percent of frames in HD World are small (<2 NALU) and more than 25 percent are larger than 10 NALU. At a mean burst length above 5 units for adaptive I-P-P-P and 2 units for fixed I-P-B-P NHK looses so many consecutive untis that the overall inversion ratio actually drops again.

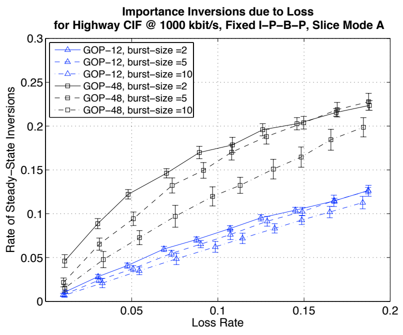
Another interesting fact is that the number of inversions rises slower than the loss rate increases, at least for streams that contain B-frames, while it is linear for streams without B-frames. The cause of this effect is that the loss probability equally affects NAL units regardless of their content. Although B-frames are often small, they contribute a significant number of data units, but their loss does not break dependency chains. Hence, at higher loss ratios more B-frame data units are lost while the number of inversions is less severely affected by their loss.



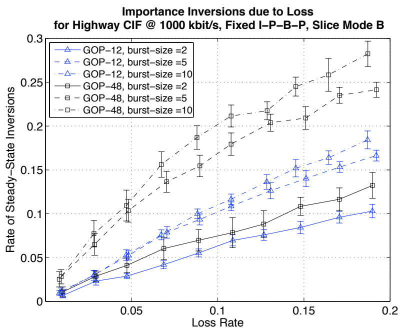
(a)



(b)



(c)



(d)

Fig. 4.29 : *Steady-state inversions for different GOP sizes in selected dependency patterns. The longer dependency chains are, the more vulnerable a sequence becomes.*

Figure 4.29 finally reveals the relation between the length of dependency chains and the number of inversions caused by loss. As expected, long dependency chains make sequences more fragile throughout all dependency patterns. Because loss bursts in slice mode A destroy multiple consecutive frames at once, the number of inversions actually decreases with increasing burst size.

4.9.5 Performance Benchmarks

When using the Dependency Validation Service in a real-world protocol or packet scheduler, the run-time performance becomes a critical issue. In this section we report on performance results we obtained from extensive measurements of a prototype implementation.

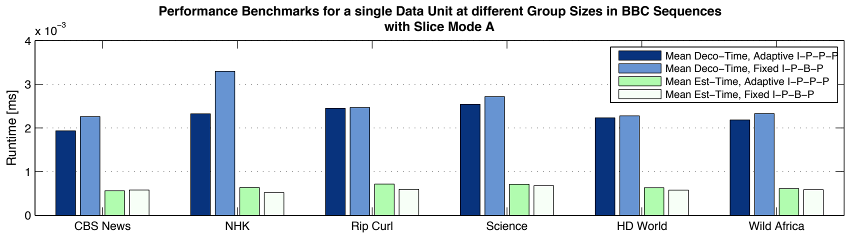
Our prototype is implemented in C++ and based on the Boost Graph Library[7]. The implementation uses adjacency lists as the central data structures to abstract graphs. In addition, STL maps are used to store type- and data unit descriptors. Maps and graphs are linked via type id's and sequence numbers to allow for quick and direct data lookup and dependency traversal while descriptors are used to cache importance values once computed. For efficient group handling the DVS does not create redundant edges for every group member. Instead the group head is the principal element for dependency control of groups.

The measurements were performed on a 64bit Intel Core2Duo processor with 2.16 GHz, 4MB 2^{nd}-level cache and 1 GB of main memory, running Darwin/MacOS X 10.4.9. The source code was compiled with GCC (version 4.0.1 for Darwin) and all optimisations were turned on. Besides the DVS performance benchmark the system ran typical workstation services, but was unloaded otherwise.

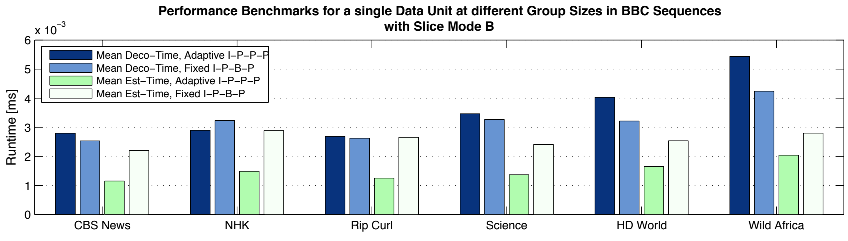
Methodology To create a reasonable measurement scenario we use the same H.264 dependency description as in all other tests and we configured the DVS as suggested in section 4.9.3. The horizon size was selected to be two times the maximal epoch size of each particular stream. For garbage collection we selected the `MaxHorizon` strategy, predictions and explicit decoration modes were enabled. As in other tests, the DVS was populated with data units during a pre-roll delay of one maximum epoch size first. We then inserted new data units step-wise, obtained the steady-state importance rating for one data unit and global object-graph metrics in every step until all data units have been processed.

During each step we measured the run-time for graph-decoration and importance estimation for a single data unit using the CPU-internal performance timestamp counter. Because this method may be influenced by CPU-scheduling and interrupt handling, we averaged the run-time over all data units in a particular stream, repeated all measurements 20 times and used the Student-t distribution to calculate confidence intervals.

[7]`http://www.boost.org/libs/graph`



(a)



(b)

Fig. 4.30 : *Decoration and estimation performance for a single data unit at multiple dependency patterns and slice modes.*

Metrics As performance metrics we use micro-benchmarks of the raw decoration and estimation run-times for a single data unit, averaged over all data units in the respective sequences. To represent the graph complexity we use the mean vertex degree, which is the average number of incoming and outgoing edges per vertex in the object graph. The average length of dependency chains is given informally only. We did not measure it because it can be analytically derived from the GOP-size and the dependency pattern. Finally, we give an analytical estimate of the combined run-time costs of dependency-based importance estimation based on a hypothetical packet scheduling mechanism.

Expectations Our implementation is optimised for estimation because we expect real users to require multiple estimations per data unit. We further expect estimation methods to scale with increased group sizes due to employed caching mechanisms. Long dependency chains, however, may have a negative impact on estimation runtime because our implementation needs to evaluate transitive references.

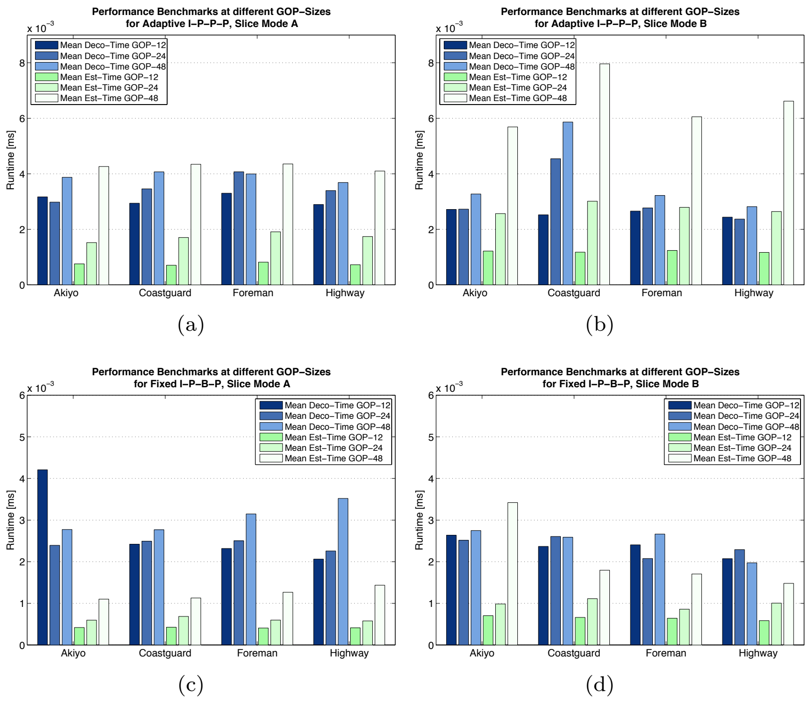


Fig. 4.31 : *Decoration and estimation performance at different GOP-sizes. While decoration scales well with increased length of dependency chains, estimation performance becomes a bottleneck.*

Results The overall performance of dependency-based importance estimation is encouraging, given the fact that other estimation models hardly perform at real-time. There is, however, one issue that require further improvement.

Figure 4.30 compares the run-time costs of graph decoration and importance estimation algorithms for different dependency patterns (fixed with B-frames and adaptive without B-frames) and different slice modes. While the selected dependency patterns seem to have no significant impact on performance, the graphs reveal that the group handling has. The dependency patterns require the DVS to create one new vertex per data unit, create several dependency relations as edges and verify them. In groups, the dependency relations of the group head are shared between all members, amortising the costs of creation. Checking relations and

		Adaptive I-P-P-P			Fixed I-P-B-P		
Slice-Mode	**Sequence Name**	**Epoch Size**	**Vertex Degree**	**DVS Run-time**	**Epoch Size**	**Vertex Degree**	**DVS Run-time**
A	CBS News	24	1.92	0.72	24	1.96	0.75
	NHK	24	1.97	0.82	24	1.71	0.70
	Rip Curl	24	1.92	0.92	24	1.71	0.77
	Science	24	1.88	0.91	24	1.68	0.88
	HD World	24	1.87	0.81	24	1.62	0.75
	Wild Africa	24	1.89	0.79	24	1.64	0.76
B	CBS News	44.68	1.29	2.70	126.45	0.73	14.27
	NHK	44.08	1.31	3.41	125.21	0.73	18.46
	Rip Curl	89.62	0.62	5.87	123.56	0.74	16.72
	Science	61.31	0.90	4.40	126.94	0.73	15.71
	HD World	121.69	0.50	13.08	312.68	0.29	45.08
	Wild Africa	121.24	0.47	10.53	326.44	0.29	42.41

Tab. 4.10 : *Performance analysis for streams with different group- and epoch-sizes using a hypothetical packet scheduler. (run-time is given in milliseconds)*

updating group members is, however, necessary for each member to reflect the correct group state. Both, decoration and estimation methods, are more expensive for group members, even though dependency relations are shared. Hence, decorating a large group in slice mode B costs considerably more than a single data unit in slice mode A. This clearly dissatisfies one of our expectations and demands further optimisations.

We are further interested in how the DVS performs at different GOP-sizes respectively dependency chain lengths. Figure 4.31 presents the measured results. As expected, the costs of estimation increase mostly linearly with the chain length in all formats. When the group-size increases too (in all slice mode B sequences), efficiency further degrades and estimation can even become more expensive than decoration.

In order to gain a more realistic view on the DVS performance we put the micro-benchmark results in relation to a hypothetical packet scheduling mechanism that operates on importance values obtained from the DVS. We are interested in the total run-time per stream per second which is required to maintain state and estimate importance values. The hypothetical scheduler runs once every 40 ms over a transmission window of two mean epoch sizes and selects an arbitrary amount of data units for transmission. The total DVS run-time t heavily depends on the mean epoch size e and the number of scheduling rounds s per second. We transmission opportunities every 40ms, $s = 25$. The run-time is calculated as $t = e \times t_{deco} + 2 \times s \times e \times t_{est}$. Results for BBC sequences in different slice modes and formats are displayed in table 4.10. The table also shows mean epoch size and mean vertex degree of the object graphs. Note, that because group members share dependency edges, the vertex degree can become less than 1 for streams with large groups. The results indicate that even the most complex HD World (I-P-B-P) stream with very a large epoch size requires less than 5 percent CPU time. Streams with smaller groups and hence smaller epochs are more efficiently handled. For example, we can estimate and schedule up to 370 instances of the CBS News stream in I-P-P-P pattern concurrently. This translates into 4 streams at one percent CPU load.

4.10 Conclusion

This chapter presented a generic and efficient framework for dependency modelling and importance estimation for packetised media streams. We introduced tools to express, verify and access structural properties of streams, and we demonstrated the efficacy of dependency-based importance estimation with real video sequences.

When adopting the dependency model, a developer should keep the following general conclusions in mind:

- Restricted visibility, either induced by loss or limited horizon size, degrades importance estimation quality.
- Inversion-free importance estimation in a limited window of data units is not guaranteed for data units until they reached the middle of the horizon.
- Strictly predictable streams do not suffer from inversions.
- Estimation in unpredictable streams relies on correct reference lists.

The proposed framework has several limitations and some issues remain for future work. First, the framework captures the importance distribution of single streams only. It provides no means to compare importance values between different streams. This becomes problematic when a scheduler operates on multiple streams with interdepending priorities, such as a combination of multiple audio and video tracks from a single application. Second, the model introduces a considerable amount of state which makes it inappropriate for lower system layers. Transport layers and proxy services are therefore the most suitable location.

The poor accuracy of our estimation model for error-resilient streams requires further improvements. We regard the following extensions as essential: (a) to model effects that emphasise or limit error propagation, such as intra-updates and skipped macroblocks, (b) to model effects of spatial coverage per data unit, and (c) to model effects of weighted prediction. Finally, it would be interesting to investigate whether dependency is useful to estimate importance in streams that were generated by encoding schemes distinct from natural video coding. Examples are wavelet coding mechanisms, texture coding, and audio encoding algorithms. We have not investigated data-partitioned, layered and scalable bitstream structures due to a lack of appropriate encoding or decoding software. We expect our dependency model to yield additional benefits here, because such streams contain more complex dependency structures and considerably more importance classes than the examined bitstreams.

Chapter Five

Noja: A Content-Aware Streaming Middleware Platform

Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem.

(David John Wheeler)

Network-adaptive streaming applications and content-aware transport layers must closely cooperate to increase the quality and robustness of multimedia delivery over best-effort packet networks. Cross-layer co-operation introduces additional complexity for application programmers and transport protocol designers and hence, we need well defined interfaces and semantics to keep complexity moderate.

In order to simplify the integration of network-adaptive streaming into applications we propose the Noja middleware platform. Noja defines general abstractions and universal communication semantics that hide the complexity of signalling and transport protocols, but expose information to coordinate the behaviour of transport protocols and applications for robust and efficient stream delivery. The main contributions of this middleware are the definition of (1) universal communication abstractions suitable for a broad range of streaming applications, (2) flexible communication semantics for diverse of application requirements and network topologies, and (3) concepts to coordinate stream delivery between network-adaptive applications and content-aware transport protocols.

In this chapter we first discuss design principles that led our development. Section 5.3 then presents Noja's communication abstractions and the design space of communication semantics for different applications. In section 5.4 we introduce implementation concepts and in section 5.5 we present example applications to show the utility of our middleware.

5.1 Design Principles

It is unlikely that a single encoding format or a single transport protocol will fit all individual requirements of streaming applications across networking technologies and user devices. Hence, the challenge is to design a simple, powerful and reusable communication middleware platform which is also format independent. The middleware should provide a single and coherent programming interface and it should allow programmers to choose the most appropriate communication semantics, transparency features and protocol stacks. The middleware should foster cooperation between adaptive applications and content-aware protocols. It should enable protocols to exploit knowledge about structure and error-resilience features of multimedia streams. It should, however, hide the complexity of transport and signalling protocols.

In the following we discuss the general principles which led the design of our middleware abstractions and architecture. We also discuss where our design differs from other multimedia platforms.

5.1.1 Assumptions on Target Environments

Noja is a communication-oriented middleware which is designed as a configurable, and isolated building block for the remote delivery of real-time media streams. Our objective is to fit Noja into existing streaming frameworks and component-based system architectures, but also to provide rich application-centric delivery services and communication semantics. We do not limit the design to special encoding formats or streaming protocols, but we omit any complex functionality that alters the semantics of a stream, such as transcoding and multiplexing. We believe that such application-level functions are ill-suited at communication layers.

We define streams as continuous, potentially indefinite, and periodic sequences of self-containing application-level data units. Because the lifetime of a stream is large and in some cases even unknown at creation time of it's first data units, we can later optimise our implementation for efficient data forwarding rather than efficient session setup. Although the session setup time may be unimportant for some applications, we note that interactive applications and large-scale streaming systems require very short setup times.

We consider several scenarios, where a middleware platform for packetised media streaming will be useful: Applications may encode their streams either live or deliver offline encoded streams. Error control may be performed at the application-level, at the transport protocol level or jointly at both levels, either sender-based, receiver-based or by a hybrid scheme. Streaming servers and clients may be separated by proxies that perform controlled content scaling, load-balancing and selective error control. We further consider that applications may or may not be sensitive to loss, may or may not require low or controlled delays, and may or may not adapt to changing network conditions.

We assume that the network randomly loses packets and also randomly delays packets that it delivers. The network may or may not support single or multiple qualities of service (predictable throughput, delay and loss). This means that variations in available bandwidth, delay and loss performance are likely to influence stream delivery when QoS mechanisms are unavailable. We further assume that the network may or may not provide a back-channel for receiver feedback, that the involved hosts may or may not have connectivity to multiple networks, and that hosts may or may not have synchronised clocks.

Hence, the middleware suites a variety of real-world scenarios of general interest, such as large-scale broadcast distribution of stored or live encoded streams, on-demand delivery of pre-encoded streams, and interactive streams over wired and wireless networks, broadcasting and multicasting networks, dedicated networks with predictable QoS, or over the best-effort Internet.

5.1.2 Interaction Model

The communication model of the Noja middleware (fig. 5.1) supports the delivery of partially loss-tolerant multimedia streams and the exchange of control and feedback signals. While data streams are delivered unidirectional, application-level signals can be exchanged bi-directionally. A *stream sender* emits a stream as a sequence of self-containing data units with a certain periodicity, while one or more *stream receivers* will receive these data units after a bounded delay with bounded jitter and loss probability. The middleware guarantees that data units are either delivered in sequence and on time or not at all. Interested applications may also monitor channel properties and subscribe to events about quality degradation and violations of service contracts.

We define that senders and receivers are loosely coupled via *explicit bindings* [10, 130], established between visible connection end-points, called *ports*. Interactions are *asynchronous* and a sender does not assume cooperation from receivers or direct reply to continue stream delivery,

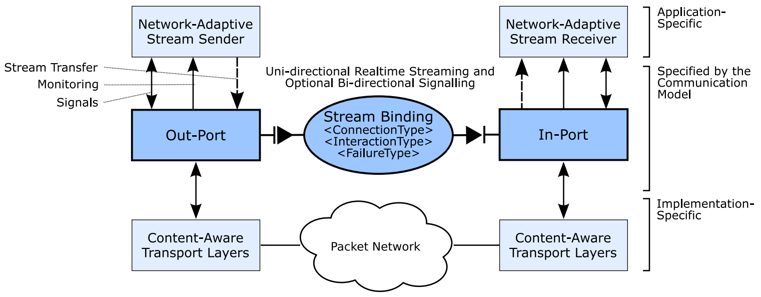

Network-Adaptive Stream Sender
Network-Adaptive Stream Receiver
Application-Specific
Stream Transfer
Monitoring
Signals
Uni-directional Realtime Streaming and Optional Bi-directional Signalling
Specified by the Communication Model
Out-Port
Stream Binding
<ConnectionType>
<InteractionType>
<FailureType>
In-Port
Implementation-Specific
Content-Aware Transport Layers
Packet Network
Content-Aware Transport Layers

Fig. 5.1 : *Components and interactions in the Noja communication model.*

because a back-channel may not be available and waiting for acknowledgements can violate deadlines of subsequent data units. When a back-channel is available, asynchronous feedback can be used at the transport protocol level to enhance robustness and also at the application level to exchange codec-control messages [172].

Data units as well as control information may get lost in transit or arrive late. While the middleware tries to compensate for such failures, the application must be prepared to either (a) wait for rebuffering, (b) receive incomplete data or (c) receive a stream termination indication. Via selectable semantics, a programmer can adjust the behaviour of interfaces to the needs of a particular application. A programmer can, for example, select quality constraints and define how operations behave in the presence of failures and receiver feedback. Network-adaptive applications can, in addition, monitor stream delivery as well as transport conditions.

Noja combines application-level framing [54] with cross-layer hints [156, 157] to enable content-awareness. When applications give proper hints to transport layers, they can experience increased efficiency and quality, while even without hints, applications benefit from Noja's convenient application-centric communication abstractions. A stream sender can attach hints as *labels* to data units. Labels contain a carefully selected set of stream properties, such as dependency, importance, timing and error-resilience information. These properties are common to all multimedia streams and independent of special encoding formats. For a detailed discussion of label contents see section 5.3.5.

Protocols in the middleware provide performance guarantees concerning delay, jitter and loss, but instead of simply controlling delay and loss they can also take the perceived quality or distortion of the reconstructed media signal at the receiver into account. Noja defines generic metadata for data units to let applications share information with streaming protocols. A protocol can use this information for selective packet scheduling and selective error control. Protocols are free to implement special optimisation models to perform these tasks.

Selective error control only yields efficiency improvements when the application tolerates limited loss of data units. Rigid applications that cannot tolerate loss or deadline violations require more expensive quality-managed channels based on resource reservations. The application programmer can request QoS delivery at session setup. The middleware then transparently performs appropriate negotiation and reservation steps.

In contrast to Remote Procedure Calls (RPC) [173] our communication model focuses on the transfer of data rather than the transfer of control. It can preserve the periodic nature of streams and ensure that data arrives in-order and in time. When loss is acceptable and timing is more important than perfect delivery, semi-reliable protocols are automatically selected and loss-events are signalled when requested. Like failure and termination models in RPCs, streams terminate on channel failure or node failure, but in contrast to the RPC model, streams can also terminate if loss exceeds an acceptable limit or when the distortion exceeds a pre-defined limit.

5.1.3 Transparency and Cross-Layer Issues

Transparency is a well known concept in communication middleware to hide effects of distribution and heterogeneity as well as implementation details of services. While transparency considerably decreases the complexity of distributed computing, hiding all communication-related issues from multimedia streaming applications is inappropriate.

Adaptive streaming applications can increase the user-perceived service quality when channel properties such as available bandwidth, delays and loss patterns are visible [78, 84, 86, 94, 96, 103, 105, 143]. Moreover, streaming applications can also benefit if lower-layer error-control and scheduling schemes, topology selection and adaptation policies are controllable by the application. Hence, our middleware design carefully balances transparency against visibility, monitoring and cross-layer coordination. Noja still provides the following forms of transparency:

Location Transparency: Ports as streaming end-points and the abstract concept of bindings effectively hide details of distribution from an application. Hence, the semantics and guarantees of operations on ports do not differ when adjacent ports are located in the same process, or in distinct processes on the same host, or on remote hosts which are connected via a local or wide-area network. Network-level names, necessary to establish signalling and transport protocol connections, are encapsulated into abstract port references in a similar way to CORBA object references. While an application needs to identify and exchange port references, their contents can remain opaque.

Failure Transparency: Noja translates common failures of distributed systems and unreliable networks into data unit loss and binding termination. Hence, permanent node and channel failures, packet loss and deadline violations are hidden behind convenient and application-centric failure classes. In addition, transient node and channel failures and packet-loss bursts are either concealed by buffering and error-control schemes or exposed as burst loss to an application. Even if no data is received, the middleware can detect outstanding data, because streaming applications expect data units with a certain periodicity.

Technology Transparency: Details of streaming transport protocols and signalling protocols are hidden behind Noja's port and binding abstractions. It remains totally transparent to an application how bindings are established and how streams are actually delivered to destination ports. A binding can, for example, choose a direct RTP protocol connection, a TCP connection or other types of media transport protocols for network transfers. The binding can also select multiple network paths or use overlay networks for cooperative streaming. When the destination is a special media processor connected to a local bus, such as a DSP or FPGA, the binding may even use different protocol stacks, hardware drivers or special data passing mechanisms.

When local systems and networks offer different service classes, the middleware abstractions allow an application to request better QoS at binding creation time. When, however, requested performance guarantees are unavailable or when established guarantees are broken, the binding is terminated and the application is informed.

Topology and Replication Transparency: Applications can express their interest to control the connection topology via binding semantics, discussed later in section 5.3.2. Bindings hide the existence of back channels from unaware applications. They can also hide the use of unicast, multicast or broadcast protocols, and the existence of multiple transmission paths. A binding can, for example, perform handover, connection replication or multipath streaming while pretending a single unicast connection to the application. Noja directly supports port- and network-level multicast. Ports do not reveal how a multicast connection is actually implemented. A binding can either use multiple unicast protocol instances, a single network-level multicast protocol, a peer-to-peer overlay network, or a mixture of these types to reach all destinations.

Instead of concealing all aspects of communication like in the RPC model, network-adaptive applications need to control and monitor communication channels. Our middleware offers the following concepts to coordinate between applications and protocol layers:

Selective Reliability: While Noja offers a simple failure model (loss and termination), it lets the application express termination conditions and the sensitivity of stream data. This information is used to adjust scheduling and error-control features of transport protocols and to trigger stream termination.

Selective Visibility and Monitoring: Binding semantics provided by Noja allow the programmer to control the visibility of connection topologies, receiver identities and receiver feedback. In addition to receiver feedback, an adaptive application may also directly monitor the channel for available bandwidth, delay and loss rates.

Selective Traffic Policing: The port abstraction offers semantics for real-time delivery and synchronisation which can reconstruct the intra-stream periodicity at the receiver and also achieve synchronised stream delivery across multiple streams and receivers. Applications can disable this feature to receive data immediately.

Selective Flow-Control: Ports let the application control the continuity of a stream so that senders can pause delivering and receivers can pause consuming a stream without interrupting the opposite peer. This is useful to signal periods of silence in a media signal and to retain protocol states when streams are stopped temporarily.

Noja supports cross-layer coordination to increase the quality and efficiency of multimedia communications. While section 4.1 already emphasised the power of hints for cross-layer system designs, we briefly introduce the set of data unit hints supported by Noja:

Data Importance: To reflect the unequal importance of data units for signal reconstruction we permit applications to attach importance and rate-distortion values. These values can guide transport protocols to adaptively choose appropriate traffic classes and selective error protection.

Data Dependency: When importance and rate-distortion values are unavailable, dependency can be used to infer dependency-based importance based on the estimation model proposed in chapter 4. In addition, protocols can use dependency relations to check data units for broken dependency after loss and selectively drop invalid units to save resources.

Deadlines: Private deadlines for each data unit are required for traffic policing and synchronisation at ports. Deadlines can also be used at lower layers to adjust scheduling and error recovery procedures. Protocols may perform retransmissions only until the deadline is reached, while scheduling schemes may prefer fresh data over older data units.

Error Resilience: Noja allows adaptive applications that can tolerate a limited amount of loss to indicate which part of their data unit payload is sensitive and which part is resilient to bit errors. A transport protocol can use this information to employ data partitioning and restrict error protection to sensitive partitions.

5.1.4 Deliberate Omissions

Noja is a pure communication-oriented middleware platform, equipped with novel interfaces to leverage cross-layer coordination. We intentionally restrict the design to the minimal necessary set of functions, such as methods to perform synchronised stream delivery, channel management, and monitoring to keep the platform simple and generic. In contrast to other multimedia middleware systems, Noja deliberately lacks the following features:

Parsing, Transcoding and Multiplexing: are typical application-level functions that require full understanding of the bitstream semantics. Transcoding and multiplexing even alter the semantics of streams. We feel that doing this in a generic communication layer is inappropriate because it breaks layering assumptions and introduces a considerable amount of complexity.

Prioritisation and Fairness between Streams: Even if Noja supports multiple receivers per stream and multiple concurrent streams per application we defer fairness and priorisation issues to streaming protocols and low-level resource allocation. Applications that require end-to-end QoS or prioritisation need to use QoS-aware scheduling and network QoS services anyway. Adaptive protocols, in contrast, are more suitable in uncontrolled networks because they perform fair rate- and congestion-control at the end-points. However, Noja provides no coordination between transport protocol end-points. Instead, Noja allows to specify resource demands at bind time and forwards these requirements to protocol implementations.

Caching and Prefetching: Scope and time-scale of stream caching and prefetching are beyond the communication-centric focus of Noja. Noja operates on sliding transmission windows, jitter compensation and synchronisation buffers which cover small sections of a stream only. In contrast, large-scale caching infrastructures replicate large stream sections up to the complete stream for load-balancing. Likewise, streaming servers prefetch large sections of popular streams to hide I/O latencies. Noja does not support these features, but caches and streaming servers may utilise Noja as transport layer.

Data Transport and Signalling Protocols: The Noja middleware architecture is extensibe. It advocates no specific streaming or signalling protocols. Hence, Noja can be customised for interoperability with existing protocol frameworks such as RTP/SIP/RTSP [5, 6, 67, 68] and emerging standards such as ISMA, 3GPP and DVB-H. The generic design allows to reuse existing libraries as transport modules. On the other hand, Noja can foster the prototyping of novel protocols and delivery services. Regardless of the actual transport and signalling modules, applications can continue to use the same abstractions and communication semantics as proposed in section 5.3.

Privacy and Security: Although an important issue, we do not consider mechanisms for privacy and security in the current design. Some of the required mechanisms can later be integrated via interceptors [174]. For example, authentication and access control easily fit into the binding setup procedure, while encryption and signature modules can be integrated into the data paths at the port layer. Other mechanisms are harder to integrate because they require extra abstractions, such as key management, security policy configuration, and privacy control. We consider these features future work and concentrate on communication and coordination tasks first.

5.2 Novelty of the Noja Middleware Platform

Advanced streaming protocols require information about the channel state and properties of the content. Coordination between different layers is currently performed in an ad-hoc manner by jointly designing and implementing applications and protocols. The high complexity of this software engineering principle leads to long development cycles, increased design and programming errors, and the decreased reusability of designs and components. Consequently, security of protocol stacks as well as the innovation of novel applications suffers.

In contrast to these joint source-channel coding approaches, Noja decouples the protocol implementation from application layers, but it retains the exchange of necessary information between them. To our knowledge, Noja defines the first generic software layer for coordination and cooperation between adaptive application and content-aware transport layers. Using the Noja architecture, protocols can be implemented once and reused by different applications for different purposes and even for different encoding formats. Noja is the first middleware design that enables the reuse of advanced streaming protocols.

The assumptions of existing middleware platforms regarding encoding formats and protocols are rather strict. Often, a single bitstream format and a single protocol is supported only. The transport protocol is either specific to the middleware or a simple version of RTP. It is well known, that RTP alone cannnot provide high-quality, robust and fair delivery of media streams because RTP lacks support for error- and congestion-control. It is also well known that a single protocol cannot support the diverse requirements of applications concearning reliability, security, mobility and efficiency. Hence, it is a natural step to provide a platform that allows multiple protocols to coexist. Noja is compatible to standard streaming protocols and it allows protocol extension and protocol evolution.

Noja's API eases the integration of stream delivery services into applications by supporting a small set of abstractions. Noja provides a single service only: the robust and efficient delivery of streams. The interfaces are easy to learn and they even remain stable across different streaming scenarios like broadcasting, on-demand and interactive streaming. A programmer can, for example, build an application on top of Noja once and later decide to exchange the protocol stack inside the middleware to fit a different network environment. Noja even allows to exchange the protocols at run-time.

The objectives of the Noja middleware architecture are similar to GOPI [130], NMM [20], Cinema [34], and DaCaPo++ [13]. Hence, the design of Noja has been influenced by these systems. Noja extends this work, however, by coordination features required to reuse advanced streaming protocols in adaptive applications. Frameworks that mostly concentrate on QoS aspects and the management of application-layer components rather than streaming protocols, such as [129], QCompiler [137], Infopipes [17], TOAST [10], and the TAO A/V Streaming Service [19] can benefit from Noja's transport services. Due to its support for QoS and monitoring capabilities, Noja is ready to be integrated into these environments.

5.3 Middleware Abstractions and Operations

In the following, we present the basic communication abstractions of the Noja middleware and show, how applications can establish connections, configure streaming semantics, exchange streams and signals, and perform monitoring.

5.3.1 Communication Abstractions

Abstractions provide a simple and consistent high-level interface for application programmers, while they hide the complexity of low-level message passing interfaces, error handling procedures, connection management and signalling protocols. For a streaming middleware we regard the following abstractions as sufficient:

Stream Units are the basic entity of communication exchanged between application modules. Stream units are typed and self-containing application-level data units, carrying payload and a *label*. This extends the well-known Application Level Framing concept [54] by a common set of metadata whose purpose is to enable content-awareness at lower layers. Stream units are independently processed at the transport level. They either arrive on time or not at all.

While the payload can contain arbitrary application data such as raw media access units (objects, frames, slices or audio samples) or preprocessed media transport units (e.g. H.264 NAL units [164]), the label has a fixed format. A label describes payload properties such as dependency, importance, timing and error-resilience information. It is passed to protocol implementation which may perform selective error control and stream scheduling.

Ports are typed communication end-points, used for data transport, stream control and monitoring. The type of a port is determined by its direction and the properties of the stream it forwards, such as media format and traffic specification. The stream direction is either IN or OUT, whereas an OUT-port is used to send a media stream towards one or more IN-ports of subsequent processing stages. Ports always handle a single stream only.

Interoperable Port References (IPR) describe and uniquely identify a port in a distributed environment similar to CORBA IORs. Besides the port type, an IPR contains signalling and location information as well as an optional set of advertised or desired transport protocols. These parameters are required for format and protocol negotiation during binding setup.

Bindings represent virtual communication channels between ports. Binding semantics are configurable at design-time, while actual performance requirements are specified at setup-time. The type of a binding determines the topology of stream delivery, the visibility of communication errors and the visibility of receiver feedback. Bindings can, for example, be local or remote, connection-oriented or connection-less, unicast or multicast. A binding hides protocol details, but exposes channel characteristics to ports in order to enable monitoring.

Synchronisation Groups coordinate the synchronised delivery of stream units at multiple and otherwise unrelated receiver ports by adding adaptive delays to stream delivery. Groups hide the details of clock synchronisation and delivery adjustment from receivers. They can support different synchronisation schemes [18,116] and group membership options.

An application can create and bind an arbitrary number of ports to combine adjacent processing stages into complex processing chains. In order to use this model in an application, a programmer must perform the following actions:

- *At design time*, appropriate classes of binding semantics must be selected to fix the interaction pattern between application stages.
- *At port creation time*, a port must be configured with stream properties such as format and traffic specifications to determine its type.
- *At bind time*, the desired performance and QoS requirements must be specified.
- *At runtime*, the application must label stream units accordingly.

Once a stream flows, an application can control the stream delivery, monitor channel conditions and optionally use receiver feedback to adapt rate and encoding to varying network and receiver conditions.

5.3.2 Binding Semantics

Our communication model draws its flexibility from an extensible set of binding types and protocol implementations. Bindings abstract from physical communication channels, hiding type and internals of the actual signalling and transport protocols used inside the middleware. Applications can only control and observe the behaviour of ports and their bindings. In order to support the diverse requirements of different streaming applications, bindings offer a set of semantics that define how ports behave in the case of communication failures and receiver feedback.

We identified three kinds of general semantics: connection semantics, interaction semantics and failure semantics. The programmer chooses the semantics at design-time, but the middleware defers the actual selection and configuration of appropriate transport protocol until run-time.

Considering the semantics, the middleware may even reconfigure and replace protocols if necessary. This gives the middleware sufficient freedom to adapt stream delivery without breaking application assumptions.

Connection Semantics Bindings generally support a single sender and multiple receivers to fork streams efficiently. In order to properly handle multiple receivers and their feedback signals, a connection semantics defines which properties of the binding's topology are visible to the sender-side application. It either exposes or hides identities of receivers, their feedback and details of the connection topology. Connection semantics directly influence the scale of stream distribution, because they define which amount of state is required at the sender side. We defined semantics for unicast, named-, hidden- and anonymous multicast connections, but we like to note that the set of semantics may be extended for special purposes. Extensions, however, require new binding objects and protocol implementations, as discussed later:

Unicast The unicast mode permits only a single receiver to be attached to the binding. It directly forwards feedback and unveils the receiver's identity to the sender. Based on the identity the sender can deny binding to a black-listed receiver, specialise the stream to the receiver and directly react to performance and error-control feedback. Unicast is suitable for small-scale applications such as personal communication systems with a point-to-point topology.

Named Multicast In named multicast mode a binding forwards a single stream to multiple receivers. The identities of all receivers are disclosed to the sender and individual feedback is delivered. This semantics combines the control and feedback options of unicast bindings with the ability to efficiently deliver a single data stream to multiple destinations. Named multicast bindings forward all feedback messages to the sender-side application. This semantics suits small-scale personal distribution systems and small multiparty conference applications.

Hidden Multicast The hidden multicast mode conceals receiver identities, but exposes aggregated feedback to the sender. This mode retains the efficiency and scalability benefits of multicast topologies while preserving the sender's ability to react to feedback. Receiver identities are either unknown to the sender-side middleware instance when network-level multicast is used, or masqueraded for application-level multicast. Hidden multicast suits mid-scale and large-scale distribution of streams in multi-hop overlay topologies and in networks with network-level multicast support.

Feedback may either arrive directly over asymmetric (unicast) back-channels or via multiple aggregation proxies. Before it is delivered to the sender application, the feedback is transparently aggregated into a single value or list of values. Examples of aggregation functions that work in multiple steps and over any number of receivers are the maximal perceived delay and loss per receiver, the minimal available bandwidth and lists of packet sequence numbers that are lost for at least a single receiver. Encoders can use this information to adjust prediction and error-resilience schemes, while transport protocols can adapt scheduling and hybrid error-control schemes.

Anonymous Multicast In anonymous multicast mode, receiver identities are unknown to the sender and feedback is unavailable or discarded. Hence, a sender can only rely on statistical models or local observations to reason about loss and delay characteristics of the channel. Without such information, the sender can only assume worst-case conditions. As a benefit, anonymous multicast is highly scalable to large receiver groups because no per-receiver state is required at the sender. Efficient implementations using connection-less broadcast channels are possible because no feedback is required. In this sense, anonymous multicast is similar to Radio and TV broadcasting systems, such as DAB, DVB and DMB and suitable to provide the same service over generic packet networks.

These semantics do not restrict implementations to a special network topology or protocol. A binding can, for example, internally use a back-channel for error control while the application requested an anonymous semantics. Hence, actual stream delivery remains completely transparent to the application. The middleware may even employ multi-path streaming, multi-protocol or overlay multicast topologies or advanced services such as network handover, load-balancing and connection replication [140].

Interaction Semantics Stream senders and stream receivers interact asynchronously, but the tightness of interactions depend on the class of the streaming application. Interactive conference systems, for example, need bi-directional low-delay interactions between a small number of known peers, while on-demand streaming systems and broadcasting applications have increasingly looser delay requirements, but also larger audiences. The desired reaction-time and the scalability demands restrict the implementation options for buffering, error control and synchronisation schemes in such scenarios. We define three main classes of interaction semantics. For every binding, a programmer can select one of these semantics:

Conversational The conversational class defines low delay as the main objective and addresses interactive applications with small numbers of known receivers. Hence, buffering and error control are tightly limited by delay, while quality degradation may be tolerated. Direct unicast or named multicast modes are preferred to authenticate receivers. Rebuffering a stream is discouraged and synchronisation usually omitted to avoid delay violations. Excessive loss usually results in immediate binding termination.

On-demand The on-demand class defines quality as the primary objective. It is designed for applications which deliver stored or live-encoded content with loose delay constraints to a mid-scale number of registered consumers. Quality and smooth delivery are more important than a low delay. Hence, start-up delays and rebuffering are acceptable and error control may reactively repair packet loss. Receiver registration and feedback require a non-anonymous connection semantics. Unacceptable loss results in rebuffering and eventually in binding termination.

Broadcast The broadcast class defines scale as its primary objective. It supports large-scale anonymous distribution of stored or live-encoded content over broadcast or overlay multicast channels to very large receiver groups. Start-up delays and rebuffering are acceptable and anonymous connection-less operation without feedback is preferred since this avoids undesired state in senders and network nodes. However, synchronisation between small receivers groups remains possible. Excessive loss results in rebuffering or termination of the receiver-side binding, while the sender side is unaffected by loss and unaware of receiver states.

Failure Semantics Failure semantics reduce the complexity of failure handling at the application level by restricting the visibility of complex communication errors to simpler models. We translate the manifold errors of distributed environments, such as link failures, node failures, message drops, missed delivery deadlines and excessive message reordering into loss errors. This is reasonable, because multimedia applications do not distinguish between these errors anyway.

At ports, stream units are either delivered entirely and in time or a loss is indicated to the receiver. However, different bitstream formats and applications are unequally sensible to loss. Hence, we define special semantics which regard the severity of errors in terms of the application and let applications express their desired degree of reliability as well as the termination condition. The design space is similar to that of RPC failure semantics, which ranges from unreliable *maybe*, through *at-most-once* and *at-least-once* to strict *exactly once* semantics.

Full reliable The full reliable class permits no gaps in the stream sequence. It translates loss into rebuffering or stream termination, thus trading predictable quality against provoked delay extensions and termination. It is useful for unprotected streams to add error resilience and for applications that require perfect delivery without loss or corruption, such as in mission-critical applications. It can, however, also be used when errors are unlikely or deterministic resource guarantees are already given by lower layers. This class is easy to implement with reliable protocols such as TCP.

Semi-reliable The semi-reliable class controls the properties of gaps in the stream sequence, such as their size and distribution, the importance, age and dependencies of lost stream units as well as conditions for rebuffering. Semi-reliability directly addresses the features of error-resilient bitstreams and partially-reliable transport mechanisms. Semi-reliability has many facets which share the common goal of optimising the trade-off between delay bounds, network efficiency and signal quality. Two useful examples are a *quality-optimised semantics*, preferring reliability over short delays, and a *delay-optimised semantics* with opposite objectives. Semi-reliability essentially requires content-aware transport protocols and loss-tolerant applications. The actual selection of optimal error-control mechanisms, protection strength and bandwidth used for redundancy is left to protocol implementations. Semi-reliable error control is most useful for unprotected streams to add a basic amount of error resilience and for applications which can trade off reliability against optimal resource consumption.

Unreliable The unreliable class permits arbitrary gaps in the stream sequence and does only terminate a stream after timeouts instead of loss patterns, while the failure semantics is that of the underlying transport protocol. This semantics is most useful when applications already perform their own error protection and require no additional protection at the transport layer. While a transport protocol inside the middleware will not add FEC and ARQ protection, it is free to use content-aware scheduling and selective drop. The advantage of this semantics is its simple implementation (on top of UDP for example). It is, however, restricted to adaptive applications that perform their own error control.

Binding Termination A binding either terminates by explicit request, when senders or receivers fail, or when an unacceptable communication error occurs. In any case, senders and receivers must establish consensus on termination to finally free acquired resources. Although interactions are asynchronous and connection-oriented protocols are optional, it is easy for receivers to detect sender or communication failures because stream units are expected to arrive periodically. Hence, at least for receivers the usual timeout mechanisms work.

Detecting a receiver or channel failure at the sender side may be impossible, in particular, when no feedback is exchanged. Thus, for correct termination of sender-side bindings, feedback is essential. The conversational and on-demand interaction semantics as well as most of the connection semantics already require feedback and back-channels. Here it is possible to send keep-alive or explicit termination messages. Broadcast interactions over anonymous multicast topologies, however, lack feedback and a back-channel. Because applications that use one of these modes do not assume any coordinated action between sender- and receiver-side bindings anyway (except real-time stream delivery), the lack of feedback for termination does not matter.

5.3.3 Port Configuration

Ports offer a variety of operation modes and configuration options to support diverse application requirements (see table 5.1 for an overview). An

Port Control Operations	
`createInPort()`	creates a receiver-side port, initial state is BUFFERING
`createOutPort()`	creates a sender-side port, initial state is HOLDING
`getIPR()`	returns a port reference
`bind()`	establishes a binding of specified type to the destination port
`unbind()`	tears down a single or all bindings of a port
Data Passing and Intra-Stream Synchronisation Operations	
`send()`	passes stream units, may block for period enforcement
`receive()`	delivers stream units, may block for period reconstruction
`wait()`	waits until next operation will not block
`waitForMultiplePorts()`	waits on multiple ports
Stream Control Operations	
`getState()`	returns current stream state
`mute()`	silently discards stream data at the receiver *(receiver only)*
`silence()`	indicates a single period of silence *(sender only)*
`hold()`	stops stream without disconnect *(sender only)*
`resume()`	(re-)starts stream after mute or hold
`purge()`	discards all data in transit *(sender only)*
`flush()`	waits until all receivers drained their channel *(sender only)*
Inter-Stream Synchronisation Operations	
`createSyncGroup()`	creates a new synchronisation group
`joinSyncGroup()`	adds the port to the group in specified mode
`leaveSyncGroup()`	removes the port from its group
Signalling Operations	
`registerSignal()`	registers a handler for the specified signal
`unregisterSignal()`	unregisters the signal handler
`raiseSignal()`	sends a signal of specified type
Monitoring and Feedback Operations	
`getTiming()`	returns execution-related timing information
`getChannelStats()`	returns channel-related statistics
`getChannelPred()`	returns short-term channel predictions

Tab. 5.1 : *Port operations for stream communication and port control.*

application can dynamically create an arbitrary number of IN- and OUT-ports according to the number and types of handled streams. To suit different execution models, ports can perform blocking and non-blocking operations. Control and monitoring operations are non-blocking.

Content-aware streaming protocols and QoS mechanisms need additional information about properties of the forwarded stream. We require applications to specify these properties at port creation time, before bindings are established and connections are set up. Because senders typically know the exact properties of a stream while receivers have at least constraints on the acceptable formats, we define an assymetric port configuration scheme. Programmers need to initialise OUT-ports with traffic and format descriptions, and IN-ports with format constraints. During binding, format description and constraints are matched and the type of the receiver-side port is completed. Contents and meaning of descriptions is as follows (see also table 5.2):

Traffic Description The traffic description defines exact or average properties of the stream's traffic pattern, such as bandwidth and buffer requirements. This is required for admission control and packet scheduling mechanisms [175,176]. Rigid applications that require QoS-provisioned services must specify their exact traffic pattern because this is used to reserve resources along the delivery path. For network-adaptive applications the traffic description may be an initial estimate on the stream properties only. Adaptive protocols can use this estimate to configure transmission windows and scheduling parameters. Period and the average interarrival time between single data units or groups of data units are mandatory. Although possible, we do not require middleware implementations to guess parameters or monitor streams to obtain them in order to keep the middleware simple and efficient.

Format Description The format description defines format-specific properties of a stream which are fixed during the lifetime of an OUT-port. The description is reliably exchanged at bind time and remains active for the connected IN-ports as long as the binding exists. The description contains a generic and globally unique encoding format ID, an arbitrary number of format-specific attributes (as key/value pairs of strings) such as video resolution or audio sampling rate, and the format-independent dependency description as defined in section 4.7.1. These properties are used to (a) check if both application-level sides agree on the format of the exchanged stream, (b) inform the receiving side of the actual stream format the sender emits, and (c) inform content-aware protocol layers of type-based dependency patterns and importance distributions. Format-specific properties are directly forwarded to the receiver side without interpretation by the middleware. Hence it is especially attractive to exchange codec parameter sets (see section 4.7.2) using this mechanism.

Format Constraints Constraints are defined for IN-ports only. They specify which stream formats are expected by the receiver-side application and which values for additional attributes are required. Attribute constraints are key/value pairs containing regular expressions. For undefined attribute keys no constraint is assumed. A constraint does only match a description when all specified attributes match.

Type	Property	Description
Blocking Mode	blocking	calls may block until resources are available
	non-blocking	calls do never block, but may indicate errors
Flow Control Mode	policed	data unit spacing is enforced by explicit wait times to shape application-level traffic
	pass-through	no enforcement, caller needs to arrange for proper spacing
Traffic Description	period	average interarrival time between data units (e.g. video frames) in ms
	maximal burst length	maximal number of data units per period (e.g. NAL units per video frame)
	average bitrate	application-level payload bitrate averaged over one second in bit/sec
	peak bitrate	maximal application-level payload bitrate per period in bit/sec
	maximum data unit size	size of the largest data unit in bit
Format Description	codec-id	name or ID of the codec (URL or FOURCC[1])
	dontFragment	indicates that data units are already properly fragmented by a network-adaptive application
	DDL specification	dependency specification generated by the DDL Compiler (see section 4.7.1)
	format-specific	content-specific properties such as frame size and encoding layers for video, audio resolutions and sampling rates, bitrates, stream length and codec parameter sets
Format Constraints	codec-id list	list of accepted codecs
	format-specific	content specific constraints (e.g. regex to match resolution, layers, and bitrate)

Tab. 5.2 : *Port properties.*

5.3.4 Binding Ports

A binding between two ports can be initiated from either side using the port interface, whereas the remote side can refuse or accept the binding. The blocking version of bind waits until a connection is fully established or the attempt timed out. A non-blocking bind returns immediately, while the connection setup is performed in the background and optionally, completion is signalled asynchronously. The required information for binding are (1) the IPR of the remote port, (2) the binding semantics, and (3) quality-of-service requirements for the connection. Ports define no means to exchange an IPR, but IPR's can be serialised and embedded into signalling protocols (e.g. SDP, SIP), exchanged via naming and location services or stored in simple files. The type of a binding is determined by connection, interaction and failure semantics which specify the expected behaviour of the respective ports. Binding semantics do also restrict the number of potential transport protocols because protocol implementations usually assume specific network topologies (e.g. available back-channels) and error-control options (e.g. loose delays for retransmissions). Not every protocol is therefore appropriate to deliver streams under every semantics.

In order to successfully establish a binding, both ports must agree on the binding, meaning that:

(a) their flow directions are opposite,

(b) the format description of the OUT-port satisfies the format constraints of the IN-port,

(c) the requested binding semantics do not conflict with an already established binding,

(d) a common transport protocol can be negotiated between both sides,

(e) the QoS requirements can be satisfied, and

(f) the remote application accepts the binding.

Conflicts under requirement (c) arise when an IN-port is already connected, an OUT-port is already member of an unicast binding or the requested binding semantics differ from the semantics of an already established binding. This effectively restricts the number of bindings for a single port to one. A single binding can, however, support multiple receivers.

An application can refuse a binding request to a port for several reasons, such as system load, failed authentication of the peer or insufficient privileges. More details are given in section 5.4.2.

Type	**Property**	**Description**
Interaction	conversational	low delay for interactive applications
Semantics	on-demand	medium delay, rebuffering and high quality for on-demand applications
	broadcast	medium delay, rebuffering and high quality for large-scale anonymous distribution
Connection	unicast	connection-oriented, single connection, known receiver
Semantics	named multicast	connection-oriented, known receivers, multiple feedback
	hidden multicast	connection-oriented, unknown receivers, aggregated, masqueraded or filtered feedback
	anonymous multicast	connection-less, unknown receivers, no feedback
Failure	full-reliable	trade perfect delivery against rebuffering or termination
Semantics	semi-reliable	controlled loss for partially loss-resilient formats
	unreliable	directly expose low-level failure model
QoS	throughput	minimal required channel throughput in bits/sec
Requirements	delay	maximal required end-to-end delay between ports in ms
	delay extension	acceptable rebuffering time in ms
	jitter	acceptable interarrival jitter in ms
	loss	acceptable data unit loss rate

Tab. 5.3 : *Binding properties.*

The QoS requirement is used for two purposes, (1) to specify performance requirements for admission control and configuration of QoS-aware channels and (2) to specify performance bounds for binding termination (see section 5.4.2 for details). QoS parameters allow the application to specify limits for bandwidth, end-to-end delay, jitter, rebuffering times, and loss. A *QoS-class* can be assigned to each parameter to separately control the severity of QoS violations. The *deterministic* class permits no violation and requires explicit resource reservations. *Statistical* services average violations over a specified interval. They require no reservations, but performance guarantees are weaker. Finally, the *best-effort* class permits any violation of the respective parameter. It only provides counts for interested applications. The QoS request and the port-specific traffic description are similar to the reserve and traffic specifications (rspec and tspec) of RSVP [9].

Bindings either terminate by explicit request, when the performance of a single quality parameter drops below an acceptable limit (according to the QoS-class) or when an unrecoverable node or connection failure is detected. After removing a binding, ports can be reused to establish new bindings even while the application still generates data units. Sending data via unbound ports results, however, in immediate loss.

5.3.5 Data Transfer and Stream Unit Labelling

Stream senders and stream receivers can exchange data streams in terms of labelled stream units via their ports. The port interface permits to send or receive exactly one stream unit per call. Stream units are delivered to receivers either entirely and on time or not at all. If a stream unit is late or (partially) lost in transit, an error is signalled to the receiver instead. Senders never receive direct loss indications from a send call because this would require the call to block until feedback is received, increasing the risk of deadline violations for subsequent data units.

Ports support blocking and non-blocking send and receive calls. A blocking receive call returns only when a new data unit was successfully received or a timeout occurred, while non-blocking receive immediately returns, either with a new data unit or an error code indicating the cause. Send calls are typically non-blocking to preserve the real-time processing of streams even if protocol queues are full. Blocking send is useful when ports are used for flow control (see sections 5.3.7 and 5.4.2 for details).

A flow-control mode determines whether subsequent send and receive calls are additionally deferred until a new stream period begins. Using the blocking mode in combination with policed flow-control, an application can be triggered by the middleware to perform actions strictly periodic. Even if sender-side queueing capacity is available or data units are waiting at the receiver-side, policed send and receive calls return only at periodic time intervals. Because streams are likely to contain multiple data units per period, ports use the timing information in data unit labels for blocking decisions. A programmer can also toggle the flow-control semantics via the port interface.

Asynchronous interactions between senders and receivers are common in distributed processing chains. Hence, send operations are non-blocking and non-policed per default, while receive operations default to blocking and policed modes.

Purpose	Attribute	Description
Dependency	`seq`	unique sequence number (mandatory)
	`type`	data unit type (optional)
	`epoch`	dependency epoch (optional)
	`enclayer`	encoding layer (optional)
	`reflayer`	referenced layer (optional)
	`short_term_reflist`	seq. of short-term references (optional)
	`long_term_reflist`	seq. of long-term references (optional)
	`is_long_term_ref`	flag to mark as long-term reference (optional)
	`group_seq`	in-group position (optional)
	`group_size`	number of data units in this group (optional)
	`imp_boost`	additional importance boost (optional)
Timing	`timestamp`	media timestamp (mandatory)
	`duration`	presentation duration for this data unit (optional)
Resilience	`coverage`	error protection coverage for non-resilient payload (optional)
	`tolerance`	acceptable loss or bit-error rate for resilient payload (optional)
	`distortion`	rate distortion value (optional)

Tab. 5.4 : *Label attributes used for cross-layer sharing of meta-data.*

Each stream unit has a unique location in the stream specified by a sequence number and a delivery deadline expressed by an associated timestamp. OUT-ports require the application to label subsequent stream units with monotonically increasing sequence numbers and with increasing delivery deadlines. While the sequence number defines the delivery order of stream units, the deadline defines the potentially periodic delivery time of a single or multiple stream units. When multiple stream units share the same deadline, a receiver can extract them from an IN-port in subsequent calls without incurring extra wait times even in blocking and policed operation modes. Policing allows to emit stream units from IN-ports exactly at their respective deadlines. With policing, a stream unit is never received too early, while deadline control guarantees that stream units are never received too late. This avoids jitter and reconstructs the exact stream periodicity, even if low-level packet scheduling and error-control introduce jitter at the packet level. Note that because jitter compensation requires buffering of stream units in IN-ports, the requested end-to-end delay must be larger than the mean network delay for compensation schemes to work effectively.

Besides mandatory sequence numbers and deadlines, labels contain optional dependency and importance information (see table 5.4 and chapter

4). Labels may also contain optional presentation duration and error resilience hints. The presentation duration can be used in protocols to calculate expected deadlines of subsequent stream units, while the error resilience information helps error-control protocols to assign additional redundancy (bits for FEC and ARQ protocols) more efficiently. The coverage attribute defines how many bits of the stream unit payload, starting at the first bit, should be error-protected, while the tolerance attribute defines how many bit errors are acceptable in the remaining, unprotected payload area.

5.3.6 Application-Level Signalling

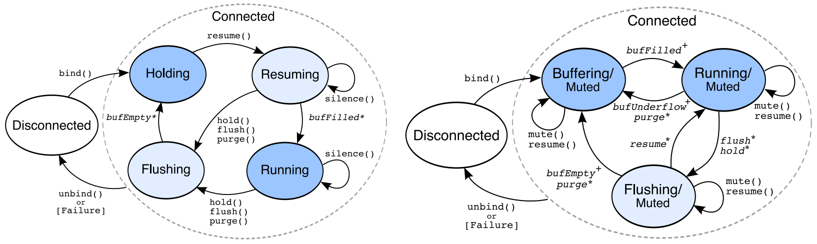
Besides stream data units, ports allow the application to exchange application-defined signals. While stream data flows uni-directional, signals can travel in both directions if supported by the binding semantics. A signal can carry control information related to the stream, such as codec control messages [172] for sender-side scaling and layer switching indications, or receiver feedback on perceived stream quality and error recovery requests (e.g. adaptive intra-refresh or reference picture selection [164]).

Signals are forwarded at the delivery guarantees of the binding. They are delivered asynchronously to registered handlers. The sender of a signal can choose between *in-order* and *urgent* delivery methods. While urgent signals are forwarded with high priority and delivered immediately, in-order signals are delivered in their local send-order (there is no partial or global order of signals between participants of a binding or across bindings). When in-order signals are sent by the stream sender, they are also synchronised with the data stream, that is, the signal is delivered to the stream receiver when the data unit preceding the signal is extracted from the IN-port.

For efficiency and complexity reasons, the type of an application-level signal is determined by a simple ID value, opaque to the middleware. The payload is restricted to a single integer value. An application is free to define reasonable ID's and payload conventions.

5.3.7 Flow-Control

Applications can control the continuity of a stream's flow (e.g. to express periods of silence) and share this information with the middleware. Streaming protocols can exploit this information for filling or draining



(a) Sender-Side States (b) Receiver-Side States

Fig. 5.2 : *State Charts for (a)* OUT*-ports and (b)* IN*-ports. (Italic names refer to local$^+$ or remote* events, while typewriter names are local operations).*

buffers, for adjusting timers and transmission windows, and for adapting synchronisation and error-control schemes. Explicit silence hints also avoid that protocols misinterpret gaps as sender or channel failures.

Due to the uni-directional flow of a stream, the stream sender usually has more control than the receiver. We allow the sender to indicate a temporary stop and later restart of the stream via `hold()` and `resume()` operations. In addition, the sender may indicate temporary periods of silence in the media signal via the `silence()` operation of its OUT-port. While a sender does not generate any data during silence periods, the receiver remains unaffected until low buffer marks are reached or the channel is drained entirely. Senders can also request to empty a channel immediately, either with the asynchronous `purge()` or with the synchronous `flush()` operation. Flush waits until all receivers processed waiting data units and signals completion to the sender. In contrast, purge immediately discards all data from protocol and port buffers and returns immediately. Flushing is useful for synchronising sender and receivers for soft stream switching, channel tear-down and soft handover procedures, while purging is required for seeking and random access to streams (e.g. in editing and interactive on-demand systems). Receivers can temporarily pause the delivery of stream units with `mute()` and restart reception with `resume()`. Data units at a muted receiver port are discarded when their deadline has passed, but the sender is not informed about muted receiver ports.

Ports maintain internal state machines to reflect the current flow-control state (see figure 5.2). When a binding exists, the sender side distinguishes between a HOLDING state with empty port and protocol buffers, a RESUMING state, where buffers are filling during a pre-roll period, a RUNNING state with filled buffers and a FLUSHING state with draining buffers. State changes are triggered by local operations with two exceptions. When the binding semantics permits a back-channel and receivers identities are known, a sender considers the buffer states of all receivers in its state machine. Here the RUNNING state reflects that all receivers are running while the HOLDING state reflects that all receivers have drained their channels and ports.

A bound receiver port distinguishes between a BUFFERING state where buffers are either empty or filling, a RUNNING state with filled buffers and a FLUSHING state with draining buffers. For simplicity we merged the respective MUTING states, where data units are not fetched by the application, as attributes into all other states. State changes at the receiver are triggered by corresponding state changes at the sender, by local observations about buffer levels and by local operations.

Initially OUT-ports start in the HOLDING state, waiting for explicit start-up, while IN-ports enter the BUFFERING state, awaiting data until a sufficient amount was prebuffered. Hold/resume cycles can trigger rebuffering at the receiver when the receive buffer became empty meanwhile. In conversational interaction semantics receiver-side ports leave BUFFERING state as soon as the maximal end-to-end delay is reached and never enter it again. In on-demand and broadcast interaction semantics sender and receivers can interact to accelerate, slow down and even stop a stream for synchronisation. Receiver-side ports leave the BUFFERING state when a high-watermark is reached and may re-enter it at buffer underflow.

5.3.8 Stream Synchronisation

Flow-control mechanisms in ports already arrange for jitter-free reconstruction of the stream periodicity, which is also called intra-stream synchronisation. To synchronise the delivery of multiple streams at a single receiver (inter-stream synchronisation) and the delivery of streams at multiple distributed receivers (inter-receiver synchronisation)additional mechanisms are required.

The Noja middleware provides the generic abstraction of *synchronisation groups* to support both forms. A synchronisation group defines policies to control the membership of ports in a group. Membership is exclusive (each port can be member of at most one synchronisation group

Property	Attributes	Description
Synchronisation Specification	policy	determines synchronisation semantics
	master mode	fixed or dynamic master election
	adaptation rate	fixed adaptation rate in ms/s (optional)
	adaptation time	maximal length of an adaptation phase in ms (optional)
	watermarks	buffer levels relative to the playout time of the first element in OUT-port buffers in ms
Synchronisation Policy	rigid	fixed target delay, no adaptation
	adaptive low delay	dynamically adapt delay to group-wide constraints, sensitive to buffer underflows
	adaptive low loss	dynamically adapt delay to group-wide constraints, sensitive to buffer overflows
Master Mode	fixed master	port remains master during the lifetime of the group
	master election	adaptive master selection based on critical buffer levels
Membership Mode	master	selected port remains master for this group (exclusive)
	slave	selected port remains slave for this group
	active member	port adapts its stream and can become master
	passive member	port only adapts its stream
	loose member	port only adapts its stream even if incompatible with the group
Watermarks	low target mark	lower limit of optimal buffer level
	high target mark	upper limit of optimal buffer level
	critical low mark	buffer underflow likely, requires group coordination
	critical high mark	buffer overflow likely, requires group coordination
	break mark	increase sender-side output delay to slow down stream
	acceleration mark	decrease sender-side output delay to accelerate stream

Tab. 5.5 : *Synchronisation properties.*

at any time), but during their lifetime ports can join and leave synchronisation groups multiple times. The actual synchronisation protocol as well as the flow adaptation mechanisms remain hidden from applications behind the port interface.

When a port is member of an adaptive synchronisation group, the port adaptively adjusts the forwarding of stream units according to a mutually agreed constraint, either stretching or compressing the inter-unit spacing. Consensus on delays and synchronisation constraints is achieved by synchronisation protocols such as [18, 116].

Table 5.5 displays the configuration options for synchronisation groups. A synchronisation policy controls when and how stream delays are adapted. The master mode and the membership options determine how a synchronisation master is selected. This can be done either statically or via dynamic election protocols which regard buffer criticality [18]. The buffer levels for triggering master switching and delay adaptation can be adjusted by means of the watermark structure.

5.3.9 Monitoring and Performance Feedback

Ports allow network-adaptive applications to obtain statistical information about their timing behaviour and the characteristics of the delivery channel.

The monitoring interface of ports provides two distinct kinds of information (see table 5.6). First, an application can observe its own execution timing which is measured by the port based on the call frequency to send or receive operations and the spacing between calls. When applications use blocking and policed send and receive operations, this timing yields exact information about the application runtime between two consecutive calls, the processing jitter, deadline misses and sleep times. This information may be utilised for adapting computational complexity of encoding algorithms.

Second, ports provide statistics on the observed channel characteristics as well as short term trends. Information include current averages for available bandwidth, loss rate, round-trip-time and one way delay as well as the MTU size along the delivery path to a selected receiver. Most of this information is only available if a back-channel exists and if the binding supports named receivers. Although this information may be obsolete and inaccurate for fast-fading wireless links, it is useful for adapting encoder error-resilience and output bitrate for many applications.

5.4 Middleware Implementation

The main objectives of the Noja middleware are a decrease in overall system complexity by proper communication abstractions that hide details of signalling and media transport protocols as well as the definition of interfaces for enabling cross-layer coordination. We implemented a prototype version of Noja in C++ which runs on several POSIX platforms,

Property	Attributes	Description
Timing Information	wait time	absolute wait time until next period starts in ms
	missed deadlines	number of missed deadlines
	call frequency	exponential moving average of observed call frequency in calls/s
	call jitter	exponential moving average of observed call jitter in ms
	load	execution time to period ratio
Channel Statistics	Path-MTU	maximum transmission unit for the total network path
	throughput	throughput observed by the receiver
	bandwidth	estimated available bandwidth on the path
	RTT	round trip time observed between sender and receiver
	FTT	forward trip time observed between sender and receiver
	PLR	packet loss rate observed by the receiver
	BER	link-layer bit-error rate (first-hop only, may be unavailable)
Channel Predictions	bandwidth	expected available bandwidth for the next period
	loss	expected packet loss probability for the next period
	bit-error	expected bit-error probability for the next period
	channel failure	expected channel failure probability for the next period
	MTTF	expected mean time to channel failure in ms

Tab. 5.6 : *Monitoring information.*

such as Linux, NetBSD and Darwin. Our prototype supports thread-based and event-driven programming models and is mainly designed for transport system tasks on top of non-realtime operation systems.

For our implementation we made extensive use of object-oriented principles and design patterns [35, 177]. Several parts of our architecture are inspired by event-driven system architectures [36, 178], zero-copy data passing schemes [179] and other multimedia platforms [13, 34, 130, 132].

5.4.1 Middleware Architecture Overview

The architecture of the Noja middleware follows a layered design, depicted in figure 5.3. Only components in the top-most layer are visible to application programmers while the other layers contain internal concepts of the middleware implementation. Besides **ports**, which are

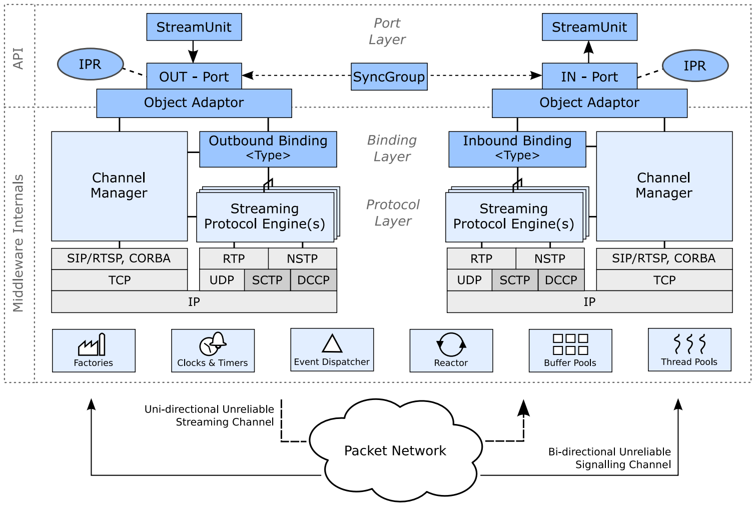


Fig. 5.3 : *Overview of the Noja middleware architecture.*

mainly used for communication, programmers can access **Object Adaptors**. Similar to CORBA object adaptors, the Noja Object Adaptor is responsible for managing the resources associated with a single media processing stage. This includes the IN- and OUT-ports of this stage, a thread pool for execution handling, a buffer pool managing the storage for stream units, as well as event and I/O dispatchers. Object adaptors offer several policies for threading (single threaded, shared thread-pool, private thread-pool), real-time execution (hard, soft, best-effort) and I/O dispatching (shared reactor, private reactor).

Central to our design is the separation between a coordinating entity, the **Channel Manager** (CM), and multiple distinct stream forwarding entities, the **Streaming Protocol Engine(s)** (SPE). While the CM is only involved in binding establishment and binding control, the SPEs independently transport streams between remote binding objects.

Because a single streaming transport protocol is not sufficient to support the great variety of binding semantics and network environments, Noja supports a pluggable SPE framework. SPEs can implement a single or multiple sets of binding semantics, such as one or more reliability classes and interaction semantics. When an application requests a new binding, a binding factory determines the most appropriate SPE for the selected combination of semantics. A binding can also use multiple SPE instances for multicasting or load-balancing and even replace SPEs at runtime to implement handover or failover schemes.

SPEs are responsible for transport and error control issues, such as fragmentation, encryption, packet scheduling, error protection, retransmissions, traffic shaping and network congestion control. Therefore a SPE can queue up stream units and transparently delay their forwarding when appropriate. In contrast, jitter compensation and adaptive playout control are performed at receiver-side ports. This is to avoid coordination of multiple receiving SPEs. A receiver-side port and respectively an in-bound binding too do only allow a single stream to be received. Multiple SPEs at in-bound bindings become necessary when failover or handover services are requested.

The concept of SPEs is not limited to special protocols such as RTP over IP networks. Our current prototype uses a light-weight transport protocol (the Noja Streaming Transport Protocol) to avoid unnecessary complexity of RTP and to add additional features more easily. SPEs based on other transport protocols (SCTP, DCCP) and special streaming standards such 3GPP and ISMA can be integrated without changing upper layer interfaces and semantics.

The *Channel Manager* is the central authority for binding establishment, signalling, and network resource management. It provides interfaces for channel setup and maintenance, QoS reservations and resource monitoring. It internally tracks existing connections, monitors network interface availability and address changes and accepts connection requests from remote nodes. When connectivity or resource availability change, the channel manager informs interested objects such as stream engines, bindings and application-level adaptation policies (through the port abstraction). If, for example, a new network becomes available, stream engines can use this network for multipath streaming or special bindings can initiate a handover procedure.

We designed the Noja architecture for interoperability and extensibility because we believe that a single protocol implementation does not fit the needs of all streaming applications. We do also not assume that all nodes in a distributed system run the Noja middleware or that all nodes belong to the same administrative domain. Hence, we designed an open architecture that allows programmers to insert new protocols without changing the semantics of the API. Noja can be enhanced in the following areas:

Bindings The implementations of binding objects can be extended for special topologies or binding semantic combinations. This allows to integrate specialised bindings later, e.g. for P2P delivery, multipath streaming support, network and session handover.

Streaming Protocols Stream Protocol Engines are the main extension mechanisms to adapt Noja to new network environments and to add new protocol functions. Due to the large amount of available error-control methods, including robust bitstream packetisation [3], packet scheduling and selective ARQ [1, 89, 143], unequal FEC [78,84,86,98,99,180] and hybrid FEC/ARQ schemes [93–96], interleaving schemes [97, 102, 103, 181], and different rate-control methods [110,111,113,182,183] the options for adjusting protocols to specific applications and network environments are manifold. Multiple protocol engines can coexist even if they implement similar semantics.

Signalling Protocols The channel manager can be extended by new signalling protocols which can even run concurrently. This allows for the inclusion of standard multimedia signalling protocols (RTSP [68], SIP [6], SDP [7]) and other front-ends such as the CORBA A/V streaming interface [19].

Synchronisation Protocols Noja allows to extend synchronisation protocols which are used inside the synchronisation group abstraction to coordinate stream playout at multiple ports. Several adaptive protocol schemes have been proposed for different environments [18,116,117,119,184]. Noja currently uses a modified version of ASP [18], described in [185].

5.4.2 Application Programming Interface

In addition to the already presented operations and semantics of our communication model (see section 5.3), we will focus on some of the implementation-level details of the Noja API in more detail. Abstractions which are visible to an application programmer are ports, object adaptors, synchronisation groups, and an efficient zero-copy buffer management system. We show how these abstractions work and how an application can use the Noja middleware under different execution models, such as thread-based execution and event-based execution.

Object Adaptor Object adaptors define an execution environment for application modules. An object adaptor manages all resources of a single processing stage of a distributed media processing chain, including IN-ports and OUT-ports, buffers and threads. The interface of object adaptors contains operations to control buffer pools, clocks, I/O policies, thread and real-time policies (see listing 5.1). These policies control how resources are allocated to applications and lower-level protocol tasks and how resources are configured (e.g. threads for real-time scheduling). Applications can request resources such as threads, buffers, clocks and timers through the object adaptor.

The threading policy controls whether the object adaptor maintains a private thread pool, shares threads from a common pool or uses a single thread for executing the application stage. The real-time policy defines if threads are scheduled with hard real-time, soft real-time or best-effort guarantees, and the I/O policy controls if a private or a shared reactor instance is used to dispatch I/O events.

Processing stages that maintain multiple IN-ports may need to wait for data to arrive on multiple ports before processing all inputs at once. This is, for example, the case for multiplexors and synchroniser components. Another example is the synchronous (re-)starting of multiple streams after ports left the (re-)buffering state. While ports already offer explicit wait calls and blocking or policed send/receive calls, these methods are limited to a single port. Waiting for multiple ports with a single thread requires additional coordination, which is performed by the `waitForMultiplePorts` operation of object adaptors. This operation waits for some or all ports in a specified set to become active. Programmers may use one of the default sets (all IN-ports, all OUT-ports, or all ports) or specify their own set of ports. They may also specify the minimal required number of active ports (0 = all) and a timeout to control when the wait operation should be aborted.

Binding When establishing a binding between ports, a programmer must provide the IPR of the remote port, select an appropriate family of binding semantics and specify the desired quality level. The IPR contains type and identity of the remote port and information on how to contact the remote channel manager to initiate a connection. When no explicit protocol is specified, a binding factory determines the most appropriate protocol which supports the desired combination of semantics and quality levels.

```
// Object Adaptor Configuration Options
enum { THREAD_SINGLE,
       THREAD_SHARED_POOL,
       THREAD_PRIVATE_POOL } ThreadPolicy_t;

enum { HARD_RT_POLICY,
       SOFT_RT_POLICY,
       BEST_EFFORT_POLICY } RTPolicy_t;

enum { SHARED_REACTOR,
       PRIVATE_REACTOR } IOPolicy_t;

typedef struct
{
    ThreadPolicy_t  thread_policy;
    RTPolicy_t      rt_policy;
    IOPolicy_t      io_policy;
} OAPolicy;

// Object Adaptor creation
ObjectAdaptor(OAPolicy policy);

// Port creation
InPort* createInPort(FormatConstraint constr);
OutPort* createOutPort(TrafficDesc tdesc, FormatDesc fdesc);

// combined wait method, default sets are {IN|OUT|ALL}_PORTS
waitForMultiplePorts(PortSet in, PortSet out, int count,
                     time_t timeout);

// Reactor access
Reactor* getReactor();

// Clock access
Clock* getClock();
void setClock(Clock* specialClock);

// Buffer-Pool access
void setBufferPool(Pool* specialPool);
Pool* getBufferPool();

// Distributed Event Service access
EventSvc* getEventService();
```

Listing 5.1: *Object Adaptor API (selected operations).*

Deterministic QoS requirements are implemented by reserving resources, while for statistical QoS requirements the channel is initially probed for throughput, delay and loss. On success, `bind` returns the ID of the new channel. This ID is required for channel monitoring and for disconnecting a distinct receiver. The `unbind` operation finally closes the channel and optionally waits until the receiver flushed all data.

```
typedef enum {DETERMINISTIC,STATISTICAL,BEST_EFFORT} CoS_t;

typedef struct {
    int   value;       // required target value
    CoS_t class;       // desired class of service
    int   percentage;  // acceptable violation per interval
    int   interval;    // averaging interval in ms
} QoS_value_t;

typedef struct {
    QoS_value_t throughput; // value in bits/second
    QoS_value_t delay;      // max. end-to-end delay in ms
    QoS_value_t delay_extension; // rebuffering in ms
    QoS_value_t timeout;    // timeout specifications in ms
    QoS_value_t jitter;     // maximal jitter in ms
    QoS_value_t loss;       // maximal packet loss
} QoS_Request_t;

typedef enum {UNICAST, NAMED_MULTICAST, HIDDEN_MULTICAST,
              ANON_MULTICAST} Connection_t;
typedef enum {CONVERSATIONAL, ONDEMAND, BROADCAST} Interaction_t;
typedef enum {RELIABLE, SEMI_RELIABLE, UNRELIABLE} Failure_t;

typedef struct {
    Connection_t  connection_type;
    Interaction_t interaction_type;
    Failure_t     failure_type;
} Binding_t;

BindingID_t bind(IN IPR_t dest, IN QoS_Request_t qos,
                 IN Binding_t family, IN Protocol_t protocol,
                 IN time_t timeout);
void unbind(IN BindingID_t bindingId, IN time_t timeout,
            IN bool flush);
```

Listing 5.2: *QoS performance specification and the* `bind()` *operation.*

Applications can express their desired service quality, reliability and termination condition a QoS specification at bind time (see listing 5.2). The interpretation of the QoS values depends on the selected class of service and the selected failure semantics. Consider, for example, a conferencing application that uses a 300kbit variable bitrate live video sequence, is tolerating loss of unimportant data units up to a limited amount, but is sensible to delay. A small amount of late data units is acceptable (they are considered lost), but rebuffering is undesired. After 10 seconds of unannounced silence, the receiver suspects sender or channel failure and requires the connection to terminate. The binding semantics in this example would be `NAMED_MULTICAST, CONVERSATIONAL, SEMI_RELIABLE`, while the QoS configuration would look as in table 5.7.

Property	Value	Class	%	Interval	Description
Throughput	$3 * 10^5$	Statistical	5	1000	average required bandwidth is 300kbit and 5 % decrease in availability is acceptable
Delay	100	Deterministic	2	1000	2 % violations per second are acceptable
Delay Ext.	0	Deterministic	0	0	no pre-buffering allowed
Timeout	10^4	Deterministic	0	0	terminate after 10 sec of silence
Jitter	10	Deterministic	2	1000	accept 10 ms jitter and 2 % violations per sec
Loss	3	Statistical	10	1000	tolerate 10 % loss below importance level 3

Tab. 5.7 : *Configuration example of QoS parameters for a video conferencing application.*

Binding establishment requires the cooperation of channel managers on both sides. They check binding semantics and QoS request for conflicts and verify the type-compatibility of the involved ports. Binding setup can optionally involve access control checks, protocol parameter negotiations, resource reservations and channel probes. If required by the connection semantics, the destination port is notified of the new binding. For binding semantics that lack connections, protocol parameter negotiation and channel reservations need to be performed in advance.

Synchronisation Groups Synchronisation groups are Noja's abstraction to enable inter-stream and inter-receiver stream synchronisation. They influence the timing and playout characteristics of IN-ports and optionally the timing characteristics of OUT-ports. Internally, a synchronisation group runs a distributed synchronisation protocol to reach consensus on a common delay constraint and master selection. We currently use a modified version of the Adaptive Synchronisation Protocol (ASP) [18] which was proposed in [185].

The synchronisation group functionality is decomposed into a single *controller* per group and a single *agent* per member port. Agents ensure that the fill-level of the stream buffer inside a port is between the specified watermarks. The identity of a group is determined by its controller. The controller performs admission control for new group members and calculates buffer target marks for all streams. Admission control ensures that a common target playout-time exists between all group members and that the watermarks of the new member agent satisfy policy constraints. In adaptive policies, the controller also elects the first mas-

```
// Controller Configuration
typedef struct
{
  uint32_t          adapt_rate; // adaptation rate in ms/s
  uint32_t          adapt_time; // time per adaptation cycle in ms
  SyncPolicy        policy;     // e.g. rigid, low_delay, low_loss
  SyncMasterMode    mode;       // e.g. fixed, election
  SyncWaterMarks    watermarks; // global buffer levels to trigger
                                // adaptations in ms
} SyncGroupSpec;

// Agent Configuration
typedef struct
{
  uint32_t        bitrate;    // stream bitrate (see traffic desc)
  uint32_t        period;     // stream period (see traffic desc)
  uint32_t        jitter      // max allowed jitter (see QoS req)
  uint32_t        skew;       // max allowed offset to group
  uint32_t        adapt_rate; // stream-specific adapt. rate in ms/s
  SyncWaterMarks watermarks; // stream-specific buffer levels to
                              // trigger adaptations in ms
  SyncMemberMode membermode; // master, slave, active, passive,
                              // loose
} SyncGroupAgentSpec;

// Controller Constructor
SyncGroupController(SyncGroupSpec spec, string controllerAddr);

// Agent Constructor (internal)
SyncGroupAgent(SyncGroupAgentSpec spec, string controllerAddr,
               Clock* clock, bool isRecvAgent);

// Port Interface for Sync-Groups
void joinSyncGroup(SyncGroupAgentSpec spec, string controllerAddr);
void leaveSyncGroup();
```

Listing 5.3: *Synchronisation interfaces.*

ter agent and confirms master switching. During adaptation, an agent controls the spacing between consecutive stream units and adaptively compresses or stretches the playout delay of the port, while delay and jitter constraints are preserved. In rigid policies, stream units are hard dropped until the buffer level returnes to the target area. When dropping does not conform to the failure semantics, the binding is terminated and the port leaves the synchronisation group.

According to the ASP protocol, which uses dynamic master/slave modes, agents exchange adaptation messages to coordinate stream delivery and master election messages to define the agent which actually controls a stream. The master agent decides how and when to adapt

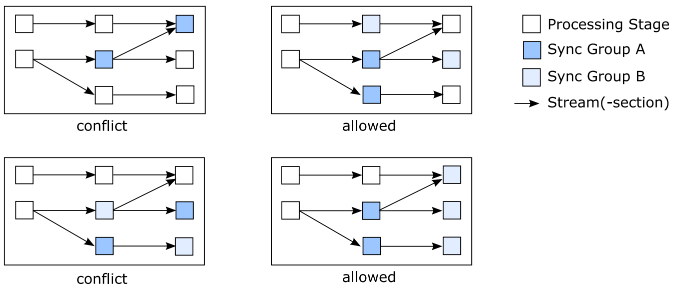


Fig. 5.4 : *Distributed synchronisation group conflicts. The squares represent processing stages of a distributed streaming application, whereas the* IN*-ports of each stage are members of a synchronisation group.*

stream delivery and propagates its decision to all group members. When a stream becomes critical, a slave agent can become master and trigger group adaptation without losing synchronisation.

Listing 5.3 shows all synchronisation related operations. Sync controllers are created without any relation to other Noja abstractions. The controller may even run on a dedicated host. The controller configuration determines the synchronisation policy which controls when and how adaptations are performed, a master selection mode which controls how the master agent is elected, and target watermarks to specify buffer control parameters (see also table 5.5). The adaptation parameter determines how aggressive a stream may be adapted. An agent may override watermarks, but the controller checks whether the new values conform to the group policy. To join a synchronisation group the port only requires an agent configuration and the serialised address (URI) of the group controller. Using the configuration, an application programmer can define a member mode to determine the role of a port in the synchronisation group and a skew value which defines the maximal allowed synchronisation offset between the stream at the local port and the group. Bitrate, period and jitter are optional, they default to the values already specified in the flow description and the QoS request. In order to allow synchronisation to an external timeline (e.g. a MIDI reference clock), a synchronisation agent can use a special clock. This clock can be set via the object adaptor interface. It is then used for all time- and timeout-related issues in ports, bindings and protocol engines attached to the object adaptor.

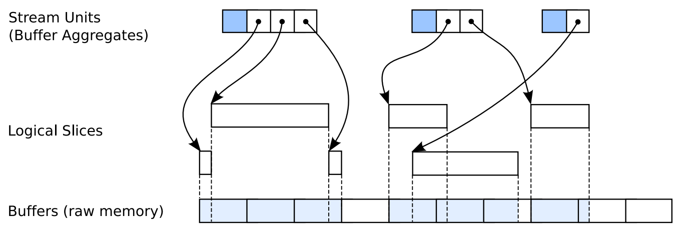


Fig. 5.5 : *Zero-copy buffers to store stream units.*

Noja synchronisation groups are generally capable of synchronising between arbitrary ports and arbitrary streams, not just IN-ports that share a communication relation to a common OUT-port. A single stream may even be synchronised multiple times (using multiple synchronisation groups) at different ports along the stream's path through the application topology. There are, however, some restriction imposed by synchronisation groups which apply to all application topologies with multiple processing stages.

Problems arise when ports join a group on which they already depend either directly or transitively. Dependency exists when (1) a single group synchronises the same stream at multiple ports of the same processing chain, and (2) when multiple groups interleave synchronisation points over distinct processing chains in different order. Figure 5.4 shows examples for conflicts and allowed synchronisation relations.

To avoid these problems, we use a mechanism to detect circular dependencies when a port joins a group. Unfortunately such mechanisms require information about the global application topology or at least knowledge about topology neighbourhood. Note that Noja's abstractions provide connectivity between adjacent stages only. A binding only relates an OUT-port of an upstream stage to one or more IN-ports of one or more downstream stages. The IN-ports and OUT-ports of a particular stage are otherwise unrelated. In other words, ports and bindings are the wrong abstraction to represent global application topology. The object adaptor, however, already relates ports of a single processing stage. Hence we integrated global topology awareness into object adaptors in the form of a topology-aware distributed event service mechanism [186]. When a binding between ports is created or destroyed, the topology information is updated.

Buffer Management Stream units are stored inside *zero-copy buffer aggregates* [179] to avoid copy overheads when caching and sharing them in multiple protocol engines. Buffer aggregates are ordered sequences of <pointer, offset, length>-pairs, called slices, where each slice refers to a contiguous section of a buffer in the virtual address space of a process (see figure 5.5).

Buffer aggregates support efficient operations for splitting, merging and duplicating data units without touching or copying the payload. Hence, stream unit fragmentation, defragmentation and multicasting are just splicing and concatenation operations of aggregate structures rather than buffer manipulations. Sharing data units between protocol engines requires no additional memory besides private aggregate structures because the actual payload buffers are shared. This makes caching and multipath forwarding of data units efficient and scalable.

Buffer aggregates are a software-based abstraction of raw memory buffers. In contrast to memory pages which are supported by the processor hardware, memory access semantics for buffer aggregates differ from conventional buffer access. This is because the payload contained in a buffer aggregate may be non-contiguously scattered in memory. Instead of obtaining a simple pointer to the complete buffer contents, buffer aggregates require the application to explicitly walk the list of buffer slices. For convenience we provide an iterator mechanisms similar to STL iterators. When a processing stage, such as an audio or video coder is unaware of the special buffer access semantics, aggregates can be serialised at the cost of an explicity memory copy operation.

When buffers are shared between aggregates, the access permissions are set to read-only to make the buffers immutable. Instead of enforcing permissions by hardware-based memory protection (the OS-kernel is not involved), an aggregate object simply refuses calls to obtain writeable pointers.

Application stages and in-bound protocol engines are required to allocate memory via a buffer pool which is managed by the object adaptor. Object adaptors can share buffer pools with other adaptors to enable zero-copy even across subsequent processing stages. Aggregates store a reference to the origin pool from which buffers were allocated to correctly return them on destruction, even when the aggregate was passed to a protocol or application module which uses a different pool for allocation.

In order to exchange special-purpose buffer pools, the object adaptor allows the application to replace the default pool. When ports local to an address space are bound, the downstream buffer pool is forwarded

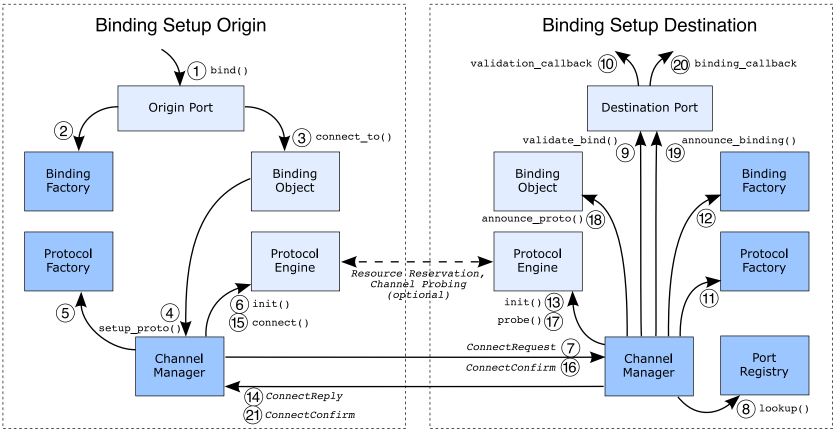


Fig. 5.6 : *Binding establishment procedure.*

upstream through the binding. Upstream application components thus can transparently allocate buffers from a pool which is controlled by a downstream component. This mechanism can increase efficiency when special memory areas such as mapped regions from hardware drivers are used. An example are rendering buffers mapped from on-board memory of graphics processors. When in-bound protocol engines receive data or upstream components generate data, it can be placed directly into consumer buffers to avoid copy operations. Copying is only required when the memory type of a buffer is incompatible to a protocol or driver, for example when a stream unit is sent forwarded to two protocol engines which use distinct memory mappings for output.

5.4.3 Binding Establishment

The abstraction of bindings is a convenient way for applications to handle connections between local or remote stream processing stages. Although applications are kept away from protocol details, the middleware must internally provide methods to select and configure bindings and protocol engines. This section discusses how Noja's generic binding setup procedure works. We show how a unicast binding is established, and discuss how the procedure must be altered for binding types that support multiple receivers and for network topologies that lack a back-channel.

For simplicity, we assume that all distributed processing stages are connected to the same network, that all stages run an instance of the Noja middleware, that all instances are equipped with a common streaming protocol engine and that the proprietary Noja-specific signalling protocol is used. Bindings can be initiated by an application from either side. Regardless of the later stream direction we will refer to the initiator of a binding as the origin and to the other side as the destination in the following.

Binding setup is atomic and hence the semantics of the binding operation are strict. Either the binding is successfully established and the stream can immediately flow or an error is reported to the caller and the application state on both sides is as if the bind operation was not called. When the bind operation returns successfully, the binding procedure ensures that the following conditions are met:

1. both sides of the applications are either informed of a new binding after the procedure completed or nothing happened on abort (does not apply to anonymous multicast)

2. the procedure either recovered from node failures and message loss or aborted

3. the destination side accepted the binding, e.g. a reject may happen at high system load or when the origin lacks permissions (does not apply to anonymous multicast)

4. the ports on both sides have agreed on the stream format, have completed the their types (note: the IN-port was unaware of the actual stream properties before), and both ports are in their initial states (the IN-port is in BUFFERING state and the OUT-port in HOLDING state)

5. the binding objects on both sides have the same type and have reached consensus on configuration parameters

6. the stream protocol engines on both sides have the same type, have reached consensus on configuration parameters and are connected (when the protocol is connection-oriented)

7. resources are reserved when deterministic service was requested or the channel performance is probed for statistical services

Figure 5.6 displays the relevant modules and interactions of Noja's internal architecture which are involved in binding setups. For a clear separation of concerns, ports only interact with their local binding object, while bindings use the Channel Manager to create and configure protocol engines. Bindings can also exchange binding-related signalling data (e.g. for hand-over procedures) via the Channel Manager. Factories are responsible for creating the proper binding types and SPE types as specified in the bind call. The Channel Manager uses signalling protocols to communicate with remote instances and a port registry at the destination to find the referenced destination port. Although signalling protocol frameworks exist, we defined a simple protocol for our prototype. Table 5.8 displays relevant message types for binding setup.

We divide the binding setup procedure into two phases, one where the channel managers perform preparations and acquires resources, and a second, were the new connection is announced to upper-layer abstractions and the user. This is similar to transaction protocols and necessary to make the binding setup atomic from the perspective of an application. The point of no return is the step where one of the channel managers announces a protocol engine upwards.

We further define a generic two-step setup procedure for protocol engines to capture any possible connection setup procedure for local or remote and connection-oriented or connection-less IPC mechanisms. During the first step (initialisation) a protocol engine is required to perform any necessary actions to initialise the communication channel, such as creating and binding sockets, opening fifos or establishing memory mappings so that an opposite protocol engine is able to immediately connect to the initialised engine. In the second step (connection) the protocol engine is asked to finally connect to the peer.

Unicast Binding Setup Without loss of generality, for this example we assume that no binding object exists in the beginning, that the origin port is an IN-port and the destination port is an OUT-port, both properly configured as described in section 5.3.3 and that the origin protocol engine will later connect to the destination protocol engine when both are initialised. This scenario is typical for a streaming client that is located behind a firewall or network address translation (NAT) gateway and wants to connect to a public streaming server.

For unicast bindings the setup procedure works as follows. In reaction to the bind call (1) and based on the requested binding semantics, the

Message Type	Property	Description/Attributes
Connection Request	request-id	monotonically increasing identifier to manage requests and detect duplicates
	epoch-id	incarnation count to detect crashes
	new_binding	requests the dest to create a new binding
	dont_connect	requests the dest to creates a passive protocol engine which is later connected from the origin side,
	dont_announce	flag to avoid the remote announcement of the new binding
	binding type	origin binding type-id
	protocol type	protocol-id either selected by the user or the channel manager
	origin IPR	IPR of the origin port, used to pass traffic and format desc.
	dest IPR	IPR passed to the bind call (to identify the remote port)
	origin binding config	binding-specific parameters (e.g. adaptation thresholds)
	origin protocol config	protocol engine-specific parameters (e.g. IP address, port, settings)
Connection Reply	status	success | failure
	request-id	id of the corresponding request
	epoch-id	epoch of the corresponding request
	failure reason	binding type not supported, protocol not supported, configuration error, rejected by application, etc
	is_connected	flag to indicate that connection setup was already performed from the destination side
	dest IPR	full IPR of the destination port, used to pass traffic and format description
	dest binding config	binding-specific parameters (e.g. adaptation thresholds)
	dest protocol config	protocol engine-specific parameters (e.g. IP address, port, settings)
Connection Confirm	status	success | failure

Tab. 5.8 : *Selected message types of the Noja signalling protocol.*

origin port first creates a new binding object via the binding factory (2) and forwards the binding request to the new object (3). Note that the bind operation expects the IPR of the remote port, a QoS request structure, the binding semantics and an optionally protocol identifier as arguments. Although the binding object knows how to forward data to a protocol engine later, it does not know how to set up a protocol engine. Hence, the binding delegates this task, including all bind arguments to the channel manager (4) and expects to retrieve a fully functional protocol engine in return. The channel manager uses the remote IPR to determine if the destination port is located on the local machine or across the network, selects a protocol engine type from the list of advertised types in the IPR (or uses the protocol-id specified in the bind call) and lets the protocol factory create the respective protocol engine (5). It then initialises the protocol engine with information from the remote IPR and the local QoS request (6). Next, the channel manger establishes a reliable control connection to the peer manager whos address is contained in the destination IPR and sends a connection setup request message (7), including setup information from the origin-side binding and from the protocol engine as well as the `dont_connect` flag because the connection should be established from the origin due to firewall restrictions.

The remote channel manager tries to find the referenced port (8), validates type and permissions of the origin port (9) and optionally requires the application to accept the binding (10). Then, the channel manager tries to create the requested protocol engine type (11) and binding type (12) using its local factories. Any failure results in a negative reply. In case of success, the channel manager initialises the destination protocol engine with traffic description, format description, QoS request data and the origin setup information (13). Because directly connecting is disallowed in this example the destination manager only replies to the origin that the destination infrastructure is in place (14), whereupon the origin protocol engine connects to the destination side (15). The reply contains an updated destination IPR as well as destination-side binding and protocol configuration data, which may be used by the protocol engine in step 15. For connection-less protocol such as UDP, connect is a no-op. Note, that without the `dont_connect` flag, the destination channel manager can decide if it connects its engine or not. This decision is signalled to the origin with the `is_connected` flag in the reply message. After successful connection setup, the origin-side protocol engine can perform resource reservations (e.g. using RSVP [9]). This conforms to the reservation direction of RSVP because in this example the origin side is the downstream receiver of the stream.

At this point, both protocol engines are configured and ready to exchange data. Now, the origin channel manager enters the second binding setup phase to announce the infrastructure upwards. It first sends a confirm message to the destination manager (16), which can optionally start probing the new channel (17). Note that regardless of the setup direction probing is always performed in downstream direction, in the example from the destination to the origin protocol engine. Note also, that probing as well as stream data transfer cannot occur until the protocol engines are connected. This is only ensured when the destination manager received the confirmation message (16). After probing completed, the destination manager announces the new engine to the binding object (18) and the binding object to the port (19). The destination port then calls an optional callback procedure to announce a new binding to the application (20). Finally, the destination manager confirms completion of the second setup phase with message (21). The origin manager then returns the ready protocol engine to the binding object which in turn signals completion to the origin port.

The additional message 21 is necessary for reaching consensus on setup phase 2. Without message 21 the origin manager could not know if the destination manager received message 16 and entered the second setup phase. The point of no return in this example is step 18, where the new protocol engine is announced to the binding. Note that because this is after any reservation and channel probing phase, the binding setup remains abortable until the requested performance guarantees are verified.

Multicast Binding Setup A multicast binding is free to use multiple unicast protocol engines, a single multicast protocol engine and even a mix of unicast and multicast protocol engines to deliver a single stream to multiple receivers. Recall that multicast is only permitted for OUT-ports, whereas an IN-port binding may use a single protocol engine only. Exceptions are binding types that perform handover and failover procedures because they may require multiple unicast protocol engines even at the receiver side. We will discuss this case later.

The different topology options for multicast delivery require changes to the previously presented binding setup procedure. In any case, the origin and the destination manager must be aware of the multicast topology and the protocol types because different interactions are required to initialise the protocols. Note that for all but the first receiver the stream already flows through the sender-side binding. Hence the setup procedure must ensure that only fully operational protocol engines are announced.

For multiple unicast protocol engines, the setup procedure changes as follows: When the origin of the binding setup is the stream sender, step (2) in figure 5.6 becomes unnecessary if the binding already exists. The remaining steps are equal to the single unicast case. When, in contrast, the origin of the binding setup is the stream receiver, the destination channel manager must obtain the binding object from the port during step (9) instead of creating a new binding object in step (12) if the binding already exists. In either case, the two-phase setup protocol ensures that a protocol engine is only announced after proper connection setup and reservations, and the stream sender receives announcements for any connected stream receiver. This directly suits the named multicast connection semantics, while for hidden and anonymous multicast, the port must filter announcements.

When the stream sender uses a network-level multicast protocol engine, no further interactions are required to add a new receiver to the sender-side protocol engine (init and connect calls are not necessary), because multicast is connection-less and the topology is controlled inside the network. The stream receiver must only create a multicast-aware protocol engine and join the appropriate multicast group. Remote interactions between channel managers are also unnecessary when a multicast address is already contained in the published IPR of the sender-side port. Such protocols are appropriate to implement the anonymous multicast semantics. In contrast, interactions between channel managers are required when the binding semantics permits receiver feedback and the receiver-side protocol engine must create a separate back-channel to meet this requirement. Although less efficient, this topology is used in some application scenarios [41].

Anonymous Multicast/Broadcast Binding Setup Broadcast bindings establish a loose coupling between ports only. Neither the sender knows about any receiver nor does a back-channel exist. In order to set up a broadcast binding, port and channel manager must perform local actions only, that is, creating a binding object, creating a broadcast protocol engine and configuring the protocol engine to either send to a channel in a broadcast network or receive data from a broadcast channel. Hence, only steps (1) – (6) in figure 5.6 are required.

Handover Bindings In contrast to multicast bindings, a handover binding may use multiple protocol engines at both sides to switch between multiple connections. It can either replicate data during handover or perform a hard switch. For either case, multiple protocol engines must be created and bound to the binding objects dynamically. Typically, there is no call to `bind()` in this case. Instead, the binding object initiates the channel setup procedure itself. The setup procedure in figure 5.6 can even accomplish this task with two minor changes. The origin binding object calls `setup_proto()` (step 4) multiple times with different configurations, but without creating a new binding object (12) or announcing to the port (19). When called in the context of `bind()`, only the first `setup_proto()` call should validate the setup (9) and create a new remote binding object (12), while only the last call should announce the destination binding object to the destination port (19). Intermediate calls should create (5/11), set up (6/15, 13/17) and announce (18, return value of 4) protocol engines only. The setup protocol offers the `new_binding` and `dont_announce` flags to enable or disable binding object creation and announcements.

5.5 Application Examples

The performance of a communication-centric streaming middleware depends in particular on the implementations of the employed protocol mechanisms. On the one hand it is difficult to present comparative performance and application-level quality measures without implementing several streaming protocol engines and resource management strategies. While related work on streaming protocols already demonstrated the utility of content-awareness in general (see chapters 2 and 3), our middleware just allows to decouple the proposed mechanisms from specific encoding formats.

On the other hand, a performance evaluation could not quantify the improvements of our novel communication model and programming environment. Hence we decided to emphasise the utility of our platform in several case-studies that show how the complexity of streaming applications decreases when using our programming abstractions. We also show, how an application can use our middleware to coordinate application-level error-control with a content-aware streaming protocol.

5.5.1 Case Studies for Coordinated Error Protection

Streaming over best-effort networks requires coordination of rate- and error-protection mechanisms to adjust them to variations in channel performance. While this task can be performed at the application level as well as in end-to-end transport protocols, a protocol-based solution has certain advantages. First it promotes the reuse of error-control mechanisms for different encoding formats and application scenarios, and second it applies even to unprotected bitstreams. In the following we present four practical scenarios to show how streaming applications can exploit our middleware platform for coordinated content-aware stream delivery.

Scenario 1: Pure Application-Level Solution The application/encoder may or may not be responsive and protect the data stream itself without requesting additional error protection from transport services. This is the design philosophy of the recent H.264 video coding standard and the RTP protocol framework, both designed for the best-effort services of the Internet. Both standards share the common concept of Network Adaptation Layer Units (NAL units) as the basic unit of transport. The efficiency and robustness of such applications may benefit from content-aware transport protocols even if this service is completely transparent. While the data streams are already properly fragmented in NAL units for robust delivery, applications still need to attach labels to signal their properties. Labels could be directly generated by an encoder or an additional NAL unit/bitstream parser. They may also be stored in hint-tracks for later reference by streaming servers.

Scenario 2: Pure Transport-Level Solution The application may or may not be responsive, but instead of protecting the data stream itself it leaves error control completely to the transport layer. Stream units contain properly labeled application-level data units, such as raw or encoded video frames. The transport layer then performs fragmentation and error protection based on the signalled data properties.

Scenario 3: Coordinated Error Protection The application is responsive and coordinates its error protection with the transport layer. It partially protects or simply prepares the bitstream for protection and leaves the actual selection of algorithms and the amount of error protection to the transport layer, which in turn keeps the application informed about the channel performance. This scheme is known as joint source-channel coding and used in integrated system designs with well known application requirements and infrastructure support, such as the voice transport in mobile phone cellular networks.

Scenario 4: Coordination for Scalable Bitstreams The application is not responsive, but generates bitstreams that are scalable and thus already contain a basic level of error resilience. The transport layer dynamically adjusts its additional protection and packet scheduling to the

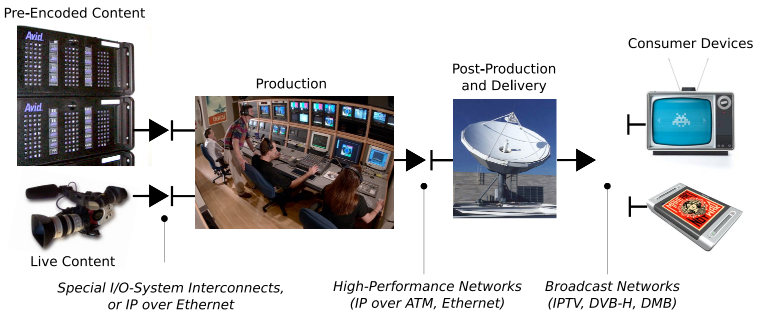

Pre-Encoded Content
Live Content
Production
Post-Production and Delivery
Consumer Devices
Special I/O-System Interconnects, or IP over Ethernet
High-Performance Networks (IP over ATM, Ethernet)
Broadcast Networks (IPTV, DVB-H, DMB)

Fig. 5.7 : *Live newscast application example: A typical broadcasting scenario with multiple processing stages, heterogeneous systems and networks, and different quality requirements.*

channel condition and the content properties, in particular, the data dependencies and importance levels. These properties must be signalled by the application. An example for this type of application is H.264/AVT Scalable Video Coding (SVC) extension [75] and its associated RTP payload format [187]. This format uses extra NAL unit types that contain the appropriate information in special header fields. A transport protocol or a network proxy can use them for unequal error protection and content scaling.

5.5.2 A Newscast Application Scenario

In this example we apply our communication model to the design of a live newscast application. While live video and audio mixing are complex application-level tasks performed by special-purpose application modules we focus on the deliver of audio-visual streams between the modules along a typical newscast production chain (fig. 5.8). We assume content is either produced live or stored in storage area networks. An audio/video mixer in the production back-end receives several live audio and video streams from cameras, inserts and synchronises prerecorded streams and outputs the resulting program for transmission to a post-production stage. This stage re-encodes the stream for delivery over a broadcasting channel, e.g. IPTV via IP multicast, RTP over an application-level multicast topology or a special-purpose DVB-H infrastructure to a large number of receivers.

This application uses two different types of network bindings (we neglect the I/O bindings for brevity), one for connecting the production back-end to the post-production stage and another for delivering streams to customers. The back-end requires high-reliability, while the customer front-end must be scalable to many receivers. Hence, the back-end uses dedicated high-performance optical networks and in the front-end the devices are connected via a broadcasting technology where a back-channel is lacking. Although the application has relaxed delay requirements, the customers should experience the least possible delay. On consumer devices buffer space is the limiting factor.

The backend-binding, connecting the production stage with the post-production transcoder uses a named multicast binding to let the application see and control connections to transcoders. Because the binding ensures receiver visibility, the sender can control that only permitted transcoders can connect. Multiple transcoders are used for fail-over, whereas each transcoder is connected to the single output binding of the production stage via reliable streaming protocols. While the protocols hide FEC and retransmission, the reliable failure semantics ensures that a deadline violation results in binding termination and becomes visible to the application. Redundant connections to other transcoders are not affected. The conversational interaction semantics allows both sides of the binding to exchange application-level signals besides the uni-directional stream.

Listing 5.4 shows a source-code example, taken from the stream output module of the production back end. It shows the configuration of port and binding as well as the send-loop of this stage. We omit the definition of the QoS parameters and error handling for clarity. In this example a dedicated thread is used to run the main loop. After creating, configuring and binding the OUT-port, the thread periodically fetches new data units from the encoder stage (line 17) which already created a correct label, waits until the next stream period starts (line 20) and sends the data unit (line 23). Internally, the outbound binding object forwards the data unit to multiple connected protocol engines for failover, while the protocol engines may use the label information to perform unequal error protection (although not necessary because a full-reliable failure semantics was selected).

The broadcast binding in the front-end is split into two local parts, one at the sender-side and one at each receiving device. The sender-side of the broadcast binding is initialised once the transcoder application starts. The anonymous binding semantics indicates that no feedback is desired, the broadcast semantics tells the protocol engines that no interaction

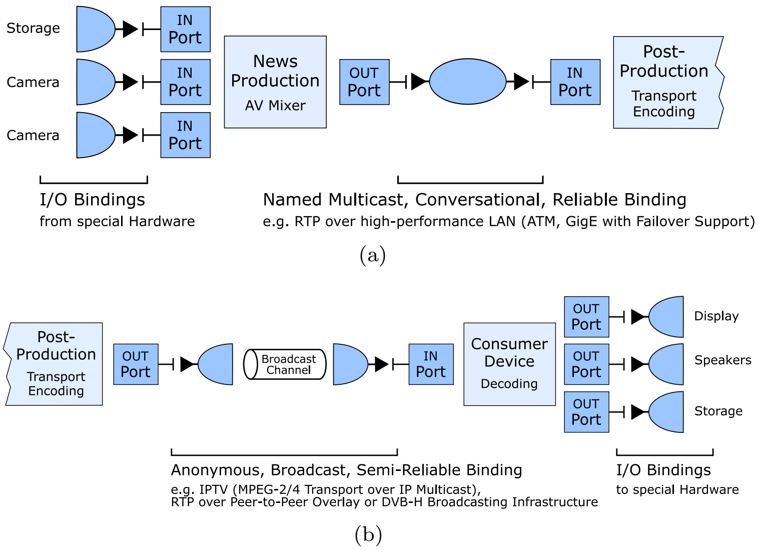


Fig. 5.8 : *Newscast example system model: Different binding types connect the distributed processing chain. While the production back-end requires reliable services and unicast delivery, the delivery frontend is anonymous and must scale to many receivers.*

with peer engines is required and that error control must be performed without feedback. The semi-reliable failure semantics requires the protocol to use unequal error protection. During binding establishment, a local protocol engine is created (e.g. either RTP, DVH-H or an IPTV protocol) and it is directed to open its side of the broadcasting channel. After successful bind, the transcoding stage periodically outputs stream data whether or not any receiver is listening. The local binding object passes the data to the protocol and the protocol transmits the data over the broadcast channel.

The receiver-side of the broadcast binding is initialised similar. However, the consumer device requires an IPR of the broadcast channel which can be installed statically. The IPR contains exact information about the channel and directs the receiver-side protocol engine to connect without further interactions. The broadcast semantics directs the receiver port to employ a startup delay and allows rebuffering when desired. The anonymous interaction semantics directs the receiver port to ignore any

```
// create and configure an out-port
OutPort_t oport = createOutPort(stream_spec);
// obtain destination IPR
IPR_t dest = lookupIPR("PostProductionIn");

// configure binding type and performance requirements
Binding_t binding_type(ANON_MULTICAST, BROADCAST, SEMI_RELIABLE);
QoS_Request_t qos_request(..);

// establish the binding
oport.bind(dest, qos_request, binding_type, 0, 0);

// send the stream periodically
while(!encoder.endOfStream())
{
    // get the next labelled stream unit
    StreamUnit_t sunit = encoder.getNextUnit();

    // wait until next stream period starts
    oport.wait();

    // and send it
    oport.send(sunit);
}
```

Listing 5.4: *Typical use-case of the port API.*

feeback the receiver tries to send, while the semi-reliable failure semantics defines that loss is generally acceptable as long as it does not exceed a threshold specified in the QoS request. As soon as data arrives over the broadcast channel, it is stored at the receiver port. The port starts emitting data when the jitter compensation buffer is filled as specified in the watermark structure.

Two receivers may in addition choose to synchronise stream delivery at their ports. This works even if the stream is already displayed at both receivers. For adaptive synchronisation, they require a bi-directional channel. One receiver must create a new synchronisation group and inform the other receiver about the address of the group controller. Then, the ports at both receivers can join the group which initially tries to find a common playout constraint. When consensus is reached, both streams are adapted until the buffer-level target area is reached.

5.6 Conclusion

This chapter presented a novel high-level programming model for multimedia streaming and the Noja middleware design that implements this model. Model and middleware support the cross-layer coordination of error-control tasks between applications and protocol stacks. Noja's main features can be subsumed as follows:

Noja is format independent: Neither programming model nor middleware interfaces or protocol implementations assume specific media encoding formats. The middleware can be used to deliver audio and video streams regardless of the bitstream features. Thus, we can support scalable streams, compressed or uncompressed btstreams, loss-resilient and loss-tolerant streams.

Noja is extensible: Our design is extensible in the following dimensions: binding semantics, binding styles, streaming protocols, signalling protocols, and synchronisation protocols. Although the proposed binding semantics already cover a large part of the design space, some applications may require additional interaction or topology models. Advanced binding styles for Peer-to-Peer distribution, multipath streaming or handover are briefly mentioned. These and other bindings can be implemented without changing the basic programming model. The same arguments apply to streaming, signalling and synchronisation protocols which are hidden from application programmers by Noja's abstractions.

Noja is interoperable: Our design does not assume that Noja runs on all nodes in a network. The Noja middleware can interoperate with standard-compliant streaming client and streaming servers as long as the streaming protocol engines follow the proposed standard for data delivery (e.g. RTP) and the channel manger complies to signalling standards (e.g. SIP, RTSP).

The proposed programming model and the middleware interfaces integrate well with different application architecture styles. This is because we provide blocking and non-blocking call semantics as well as synchronous and event-driven execution styles. Buffer management, clock implementations, socket handling and I/O event demultiplexing can also be adapted and integrated into different frameworks.

Noja is adaptive: Programmers can adapt our platform to application requirements at design-time by selecting appropriate binding semantics and the interface call model. During session setup, the application can influence the protocol selection process and define QoS requirements. A binding is free to replace protocols and channels at run-time. Finally, the monitoring and event callback features enable programmers to adapt an application at run-time.

The Noja middleware hides details of session management, streaming protocol negotiation and implementation artefacts. During application design, a programmer can select the desired error model, interaction model and topology features. At runtime, the programmer can control quality parameters and he can use channel and receiver feedback to adapt a stream to varying network conditions.

However, application programmers need to pre-fragment bitstreams into data units and provide the necessary meta-information about these data units. The costs of obtaining this information can be low when hint tracks are used, but the can also be higher, in particular, when media encoders have no interfaces to export metadata.

These features enable Noja to be applicable in many application scenarios and in diverse application architectures and runtime environments. Using Noja, programmers can implement applications once and port them to different network at with low implementation and configuration efforts (changing a default protocol for a semantics requires changes in the protocol factory configuration only). This feature does also allow researchers and engineers to quickly transfer a complete application to a network simulator for analysing and comparing streaming protocols.

So far, Noja is still in the design stage. In order to prove the efficiency and usefullness of the programming model the middleware as well as several protocols and different applications must be implemented. This is necessary to measure the run-time overheads of the implementation and to provide evidence of the applicability of the proposed programming abstractions and middleware interfaces.

Chapter Six

Conclusion and Future Work

This 'telephone' has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.

(Western Union internal memo, 1876)

While there is a consensus in research and industry that future multimedia systems will be layered on top of best-effort IP networks, engineering proper communication mechanisms remains challenging. Variations in delay, loss and available bandwidth will continue to afflict applications and transport protocols. Adaptive solutions for error-control and rate-control such as adaptive unequal error protection, selective retransmissions, and hybrid schemes prove to be the optimal approaches in such environments. Because they require intimate knowledge about the semantics of a bitstream, current approaches are often tied to special encoding formats. This effectively reduces their reusability and it also increases the overall system complexity. In response to these observations, we believe that a generic and content-aware middleware layer is an appropriate solution.

In this thesis, we identified meta-data that describes properties of data units in media streams and we proposed a content-awareness framework that enables system layers to access and track these properties in a generic way. We also proposed the Noja programming model and its implementation. Noja is built around basic communication abstractions, namely ports, bindings and stream units, as well as three sets of communication semantics to control interaction options, the visibility of receivers and their feedback and the visibility of failures. Noja can selectively hide the complexity of streaming and signalling protocols from applications, but it does also enable applications to share content-specific data properties with transport layers and obtain feedback about the delivery channel.

We showed that this small set of abstrations is powerful enough to support a broad range of different streaming applications with diverse quality and reliability requirements. The middleware allows the reuse of advanced streaming protocols in different application scenarios and has the potential to add error-resilience even to unprepared media streams.

6.1 Achievements

The two main contributions of this thesis are our *Dependency Model* and the *Noja Programming Model*. In combination, both achieve the following improvements:

Reduction in Complexity: The middleware interfaces completely hide streaming and signalling protocol implementations. Programmers only need to work with a small set of abstractions (Port, Binding, IPR, Stream Unit and Synchronisation Group). These abstractions do already implement the necessary functionality to establish streaming sessions and to deliver multimedia bitstreams in a robust, timely, and synchronised way, regardless of the bitstream format. Programmers are only required to pre-fragment a stream and attach meta-data to the exchanged data units. This effectively reduces the application complexity because a programmer must no longer deal with buffer management, jitter-control, and error-control which is often lacking in todays streaming protocol libraries.

Cross-Layer Control and Adaptation: Both, our dependency model and our middlweare concepts aid adaptation and coordination of error-control tasks across system layers. The middleware itself can be adapted to application requirements at design-time by selecting communication semantics and at run-time by specifying protocol and QoS requests. Using the dependency model, an advanced protocol can directly adapt its scheduling and selective error-control to the transported content. The Noja middleware provides interfaces for the cross-layer exchange of meta-data and monitoring feedback. This effectively enables the implementation application-level adaptation mechanisms. Note also that this cross-layer design does not increase the overall system complexity because layers are still independent and cross-layer meta-data is exchanged as hints only.

Flexibility: Noja is format independent, extensible, and interoperable by design. Neither programming model nor middleware interfaces or protocol implementations assume specific media encoding formats. The programming abstractions support diverse application topologies and interaction patterns. Programmers can easily add new binding semantics and binding styles, as well as new streaming, signalling and synchronisation protocols. Our design does not assume that every node in a network runs the Noja middleware. Instead, Noja can interoperate with standard-compliant streaming client and streaming servers by means of standard protocol implementations.

Due to its flexible interfaces Noja integrates well with different architectural styles, buffer management systems, and I/O event demultiplexing systems. We also believe that the inclusion of privacy and security architectures below the port interface can be done without changing applications or large parts of the middleware.

6.2 Open Questions

The proposed dependency framework still has some limitations. It provides no means to compare importance values between different streams and it suffers from unpredictability when meta-data is lost. It is especially interesting to investigate the limitedly-predictable class of streams to find static or dynamic properties that may improve prediction. Static properties can be included into the type description for a specific bistream format (e.g. a sub-profile of a standard such as H.264). Dynamic properties can be signalled in-band or out-of-band when the stream flows and the type-graph can be updated when updates are available. We did not investigate the power and the costs of dynamic type-graph updates.

The accuracy of the estimation model for error-resilient streams should be improved. The model may use additional information about encoder error-control features, such as intra-updates, skipped macroblocks, flexible macroblock ordering and weighted prediction.

We did also not analyse the importance estimation accuracy of the dependency model for data-partitioned, layered and scalable bitstream structures. We believe that the model has a lagre potential, in particular, when used with scalable bitstreams. Extending the dependency model to other encoding mechanisms (wavelet coding) and other domains (audio and texture coding) remains an open issue.

Although the Noja programming model is thoroughly described in this thesis, the implementation of all middleware features is still incomplete. In order to prove the efficiency and usefullness of the programming model the middleware as well as several protocols and different applications must be implemented. This is necessary to measure the run-time overheads of the implementation and to provide evidence of the applicability of the proposed programming abstractions and middleware interfaces.

Another open question is, whether the extraction of content attributes at the application level and the use of these attributes at a lower protocol layers is more efficient or more robust at all. A fully functional cross-layer system needs therefore be compared with solutions that solely operate on the application level.

6.3 Future Research Directions

The presented contributions in this thesis followed a path to investigate potential benefits of cross-layer coordination. We believe that cross-layer designs can improve the efficiency and robustness of system architectures. In this work, we focused on the multimedia streaming domain because we expected room for optimisations due to the high resource demands, realtime requirements and the partial loss tolerance. While this thesis just developed architecture support we still need existing (rate-distortion) optimisation models to prove the actual benefits of cross-layering. It would be interesting to see how a streaming protocol performs when it just uses dependency information, instead of full distortion values.

It is still unclear whether and how cross-layer designs should be integrated into general purpose system architectures and, in particular, general purpose Internet protocol stacks. In special-purpose protocol stacks (e.g. 3GPP for mobile wireless networks) cross-layer designs are already used successfully. However, the engineering of such protocol stacks is done by small developer groups and the devices operate in a controlled environment. The fast evolution of 3GPP standards usually restricts the evolution and the lifetime of such protocols to a couple of years.

This thesis designed a special flavour of cross-layer coordination, which uses optional hints and monitoring-events to avoid breaking layer assumptions. Future research should investigate whether this is a useful design pattern or if alternative cross-layer designs are more efficient or can provide better features. It is of particular interest whether cross-layer ideas are applicable to areas besides communication, such as storage systems, virtual machines, memory allocation and CPU scheduling.

We just analysed how hints should be structured for multimedia streaming systems in order to express content-specific attributes. Although the thesis did not define a special content-aware streaming protocol, the proposed middleware design supports the main approaches from the rate-distortion streaming literature and beyond. However, we did not consider passing hints to lower protocol layers, such as the link-layer.

Appendix A

Dependency Description Language

```
DDL = TYPE_DECLARATION  DEPENDENCY_DECLARATION;

TYPE_DECLARATION = 'types' '='  '{' TYPE ';' { TYPE ';' } '}' ;

TYPE = TYPE_NAME '(' { TYPE_PARAM ',' } TYPE_PARAM ')' ;

TYPE_NAME = { LETTER | NUMBER | '_' } ;

TYPE_PARAM = (( 'avg_imp'  | 'min_deps' | 'max_deps' | 'min_imp' |
                'prediction_burst' | 'prediction_offset' |
                'prediction_period' ) '=' INTEGER ) |
              ( 'group_semantic' '=' ( 'no' |'equal' | 'unequal' |
                                     'refinement' ) |
              ( 'starts_epoch' '=' ('true' | 'false'))
             ) ;

DEPENDENCY_DECLARATION = 'dependency' '(' TYPE_NAME ')' '='
                         '{' RELATION ';' { RELATION ';' } '}' ;

RELATION = SELECTOR '(' RELATION_PARAMS ')' KIND ;
SELECTOR = 'last_of' | 'all_of' ;
KIND = 'weak' | 'strong' ;

RELATION_PARAMS = TYPE_NAME [ ',' DISTANCE
                            [ ',' EPOCH_DISTANCE
                            [ ',' LAYER_DISTANCE
                            [ ',' EPOCH_INTERLEAVING ]]]] ;

DISTANCE = INTEGER ;
EPOCH_DISTANCE = INTEGER ;
LAYER_DISTANCE = INTEGER ;
EPOCH_INTERLEAVING = INTEGER ;

LETTER = 'A'..'Z' | 'a'..'z' ;
NUMBER = '0'..'9' ;
INTEGER = NUMBER { NUMBER } ;
```

Listing A.1: *EBNF of the Dependency Description Language.*

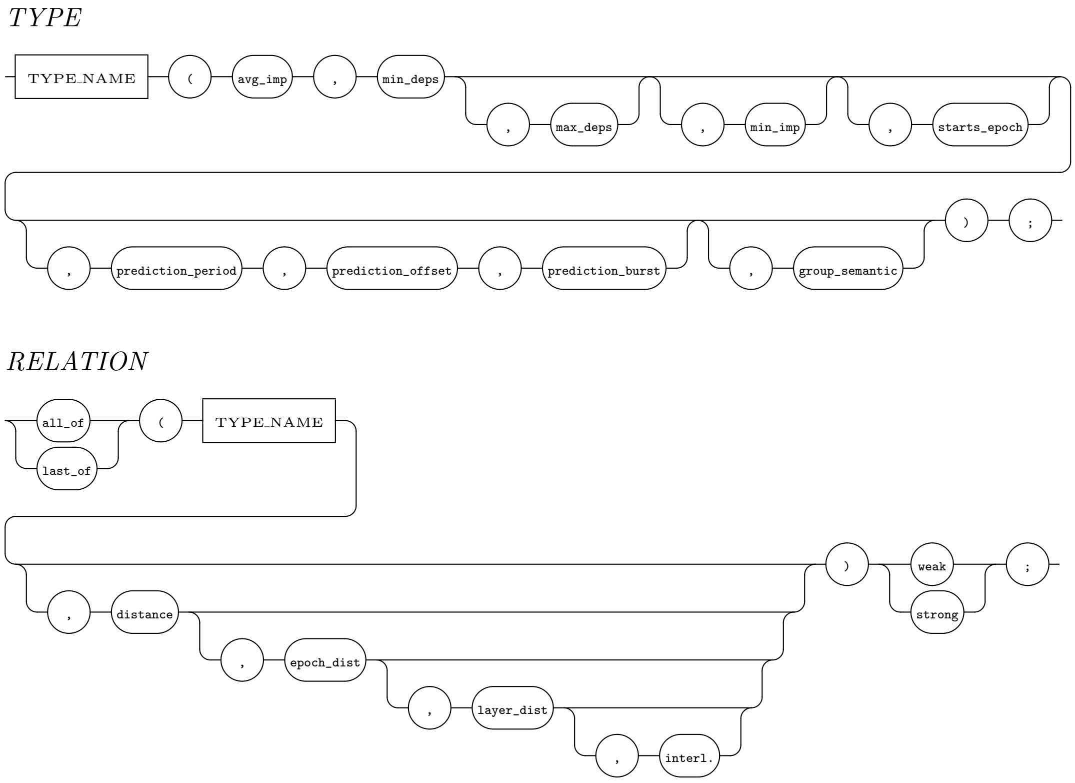


Fig. A.1 : *Syntax diagrams for a a single type specification and for a single dependency relation specification.*

```
#  DDL Description for General H.264 Sequences
#
#

types = {
  # Parameter Sets
  NALU_7_SPS( avg_imp = 255, min_deps = 0 );
  NALU_15_SPS_SUB( avg_imp = 255, min_deps = 1 );
  NALU_13_SPS_EXT( avg_imp = 254, min_deps = 1 );
  NALU_8_PPS( avg_imp = 253, min_deps = 1 );
  NALU_14_PREFIX( avg_imp = 252, min_deps = 1 );

  # Frame-data Containers
  NALU_5_IDR( avg_imp = 6, min_deps = 2, group_semantic = equal );
  NALU_1_NON_IDR( avg_imp = 5, min_deps = 2, group_semantic = equal );
  NALU_19_AUX( avg_imp = 3, min_deps = 2, group_semantic = equal );
  NALU_20_SVC( avg_imp = 4, min_deps = 2, group_semantic = unequal );

  # Data Partitioning
  NALU_2_DPA( avg_imp = 5, min_deps = 2, group_semantic = unequal );
  NALU_3_DPB( avg_imp = 4, min_deps = 3, group_semantic = unequal );
  NALU_4_DPC( avg_imp = 3, min_deps = 3, group_semantic = unequal );

  # Extra Data Containers
  NALU_6_SEI( avg_imp = 1, min_deps = 0 );
  NALU_9_AUD( avg_imp = 2, min_deps = 0 );
  NALU_10_EOSQ( avg_imp = 2, min_deps = 0 );
  NALU_11_EOS( avg_imp = 2, min_deps = 0 );
  NALU_12_FILL( avg_imp = 2, min_deps = 0 );
}

dependency(NALU_13_SPS_EXT) = {
  last_of(NALU_7_SPS, 1, 2) strong;
}

dependency(NALU_8_PPS) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_15_SPS_SUB, 1, 2) weak;
}

dependency(NALU_5_IDR) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
}

dependency(NALU_1_NON_IDR) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_1_NON_IDR) weak;
}

# continued on page 220
```

Listing A.2: *Full H.264 dependency description.*

```
#  continued from page 219

dependency(NALU_19_AUX) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_1_NON_IDR) weak;
}

dependency(NALU_20_SVC) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_15_SPS_SUB, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_14_PREFIX, 1, 2) weak;
}

dependency(NALU_2_DPA) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_2_DPA) weak;
}

dependency(NALU_3_DPB) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_2_DPA) weak;
}

dependency(NALU_4_DPC) = {
  last_of(NALU_7_SPS, 1, 2) strong;
  last_of(NALU_13_SPS_EXT, 1, 2) weak;
  last_of(NALU_8_PPS, 1, 2) strong;
  last_of(NALU_5_IDR) weak;
  last_of(NALU_2_DPA) weak;
}
```

Listing A.3: *Full H.264 dependency description (continued).*

Bibliography

[1] P. Chou and Z. Miao. Rate-distortion optimized streaming of packetized media. *IEEE Trans. on Multimedia*, 8(2):390–404, 2006.

[2] B. Girod, J. Chakareski, M. Kalman, Y. J. Liang, E. Setton, and R. Zhang. Advances in network-adaptive video streaming. In *Proc. of the International Workshop on Digital Communications (IWDC 2002)*, pages 1–8, Sep 2002.

[3] M. Etoh and T. Yoshimura. Advances in Wireless Video Delivery. *Proc. of the IEEE*, 93(1):111–122, Jan 2005.

[4] V. Srivastava and M. Motani. Cross-Layer Design: A Survey and the Road Ahead. *Communications Magazine, IEEE*, 43(12):112–119, 2005.

[5] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[6] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916.

[7] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), July 2006.

[8] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006.

[9] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. RFC 2205 (Proposed Standard), September 1997. Updated by RFCs 2750, 3936, 4495.

[10] Tom Fitzpatrick et al. Design and Application of TOAST: an Adaptive Distributed Multimedia Middleware Platform. In *8th International Workshop on Interactive Distributed Multimedia Systems*, volume 2158 of *LNCS*, pages 111–123, Lancaster, UK, 2001.

[11] Geoff Coulson. A Configurable Multimedia Middleware Platform. *IEEE MultiMedia*, 6(1):62–76, 1999.

[12] Denise J. Ecklund, Vera Goebel, Thomas Plagemann, and Jr. Earl F. Ecklund. Dynamic end-to-end QoS management middleware for distributed multimedia systems. *ACM Multimedia Systems*, 8(5):431–442, 2002.

[13] Burkhard Stiller, Christina Class, Marcel Waldvogel, Germano Caronni, and Daniel Bauer. A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience. *IEEE Journal on Selected Areas in Communications*, 17(9):1580–1598, Sep 1999.

[14] David A. Karr, Craig Rodrigues, Joseph P. Loyall, Richard E. Schantz, Yamuna Krishnamurthy, Irfan Pyarali, and Douglas C. Schmidt. Application of the QuO Quality-of-Service Framework to a Distributed Video Application. In *International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.

[15] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, and David E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. In *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.

[16] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The Design and Performance of a Real-Time I/O Subsystem. In *IEEE Real Time Technology and Applications Symposium*, pages 154–163, 1999.

[17] Andrew P. Black, Jie Huang, Rainer Koster, Jonathan Walpole, and Calton Pu. Infopipes: an Abstraction for Multimedia Streaming. *ACM Multimedia Systems Journal*, 8(5):406–419, 2002.

[18] Kurt Rothermel and Tobias Helbig. An adaptive protocol for synchronizing media streams. *Multimedia Systems*, 5(5):324–336, 1997.

[19] Sumedh Mungee, Nagarajan Surendran, Douglas C. Schmidt, and Yamuna Krishnamurth. The Design and Performance of a CORBA Audio/Video Streaming Service. In *32. Hawaiian International Conference on System Sciences*, 1999.

[20] Marco Lohse and Michael Repplinger and Philipp Slusallek. An Open Middleware Architecture for Network-Integrated Multimedia. In *Workshop on Interactive Distributed Multimedia Systems*, volume 2515 of *LNCS*, pages 327–338, 2002.

[21] Qian Zhang, Wenwu Zhu, and Ya-Qin Zhang. End-to-end QoS for Video Delivery over Wireless Internet. *Proc. of the IEEE*, 93(1):123–134, 2005.

[22] B. Girod, M. Kalman, Y. Liang, and R. Zhang. Advances in channel-adaptive video streaming. In *International Conference on Image Processing (IWDC 2002)*, pages I–9 – I–12, 2002.

[23] John Apostolopoulos, Wai-Tian Tan, and Susie Wee. Video Streaming: Concepts, Algorithms, and Systems. Technical Report HPL-2002-260, Hewlett-Packard Laboratories, Palo Alto, California, 2002.

[24] Dapeng Wu, Y.T. Hou, Wenwu Zhu, Ya-Qin Zhang, and J. M. Peha. Streaming Video over the Internet: Approaches and Directions. *IEEE Trans. on Circuits and Systems for Video Technology*, 11(3):282–300, 2001.

[25] T. Plagemann, V. Goebel, P. Halvorsen, and O. Anshus. Operating System Support for Multimedia Systems. *The Computer Communications Journal, Elsevier*, 23(3):267–289, 2000.

[26] G. Carle and E. Biersack. Survey of Error Recovery Techniques for IP-based Audio-Visual Multicast Applications. *IEEE Network*, 11(6):24–36, Dec 1997.

[27] Thomas Wiegand, Gary J. Sullivan, Gisle Bjntegaard, and Ajay Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[28] Ian E. G. Richardson. *H.264 and MPEG-4 Video Compression – Video Coding for Next-generation Multimedia.* Wiley, 2004.

[29] European Telecommunication Standards Institute (ETSI). ETS 300 401. Radio broadcasting systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers. Second Edition. Available from `http://www.etsi.org/` as ETS 300 401, May 1997.

[30] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky. A Transport Layer for Live Streaming in a Content Delivery Network. *Proc. of the IEEE*, 92(9):1408–1419, 2004.

[31] Thorsten Strufe. *Ein Peer-to-Peer-basierter Ansatz für die Live-Übertragung multimedialer Daten.* PhD thesis, TU Ilmenau, 2007.

[32] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *SOSP'03*, 2003.

[33] Jens-Rainer Ohm. Advances in Scalable Video Coding. *Proc. of the IEEE*, 93(1):42–56, 2005.

[34] Kurt Rothermel, Ingo Barth, and Tobias Helbig. Cinema - An Architecture for Distributed Multimedia Applications. In *Int. Workshop Architecture and Protocols for High-Speed Networks*, pages 253–271, 1993.

[35] F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* John Wiley and Sons Ltd, 1996.

[36] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[37] H. O. Rafaelsen and F. Eliassen. Trading and Negotiating Stream Bindings. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 289–307, 2000.

[38] Mohammed Ghanbari. *Standard Codecs: Image Compression to Advanced Video Voding.* IEE Telecommunications Series. Institution of Electrical Engineers, London, 2003.

[39] J. Ribas-Corbera and P. A. Chou. A Generalized Hypothetical Reference Decoder for H.264/AVC. *IEEE Trans. on Circuits and Systems for Video Technology*, 13(7):674–687, 2003.

[40] Hang Liu, Hairuo Ma, Magda El Zarki, and Sanjay Gupta. Error Control Schemes for Networks: An Overview. *Mobile Networks and Applications*, 2(2):167–182, 1997.

[41] M. Westerlund and S. Wenger. RTP Topologies. RFC 5117 (Informational), January 2008.

[42] Asfandyar Qureshi, Jennifer Carlisle, and John Guttag. Tavarua: Video Streaming with WWAN Striping. In *ACM Multimedia 2006*, Santa Barbara, CA, October 2006.

[43] Paolo Bellavista, Antonio Corradi, and Luca Foschini. Application-level Middleware to Proactively Manage Handoff in Wireless Internet Multimedia. In *8th International Conference on Management of Multimedia Networks and Services (MMNS'05)*, pages 156–167, 2005.

[44] Fernardo Pereira and Touradj Ebrahimi. *The MPEG-4 Book*. Prentice Hall, 2002.

[45] J.-F. Huard, A. A. Lazar, Koon-Seng Lim, and G. S. Tselikis. Realizing the MPEG-4 Multimedia Delivery Framework. *IEEE Network Magazine*, 12(6):35–45, 1998.

[46] D. Wu, Y. Hou, W. Zhu, H. Lee, T. Chiang, Y. Zhang, and H. Chao. On End-to-End Architecture for Transporting MPEG-4 Video Over the Internet. *IEEE Trans. on Circuits and Systems for Video Technology*, 10(6):923–941, Sep 2000.

[47] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar. RTP Payload Format for MPEG1/MPEG2 Video. RFC 2250 (Proposed Standard), January 1998.

[48] J. van der Meer, D. Mackie, V. Swaminathan, D. Singer, and P. Gentric. RTP Payload Format for Transport of MPEG-4 Elementary Streams. RFC 3640 (Proposed Standard), November 2003.

[49] Stephan Wenger. H.26L over IP: The IP-Network Adaptation Layer. In *Proc. of 11th International Packet Video Workshop (PV2002)*, Apr 2006.

[50] S. Wenger, M.M. Hannuksela, T. Stockhammer, M. Westerlund, and D. Singer. RTP Payload Format for H.264 Video. RFC 3984 (Proposed Standard), February 2005.

[51] Ye-Kui Wang, M.M. Hannuksela, S. Pateux, A. Eleftheriadis, and S. Wenger. System and Transport Interface of SVC. *IEEE Trans. on Circuits and Systems for Video Technology*, 17(9):1149–1163, 2007.

[52] S. Wenger, Wang Ye-Kui, and T. Schierl. Transport and Signaling of SVC in IP Networks. *IEEE Trans. on Circuits and Systems for Video Technology*, 17(9):1164–1173, 2007.

[53] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[54] David D. Clark and David L. Tennenhouse. Architectural Consideration for a New Generation of Protocols. In *Proc. of the 1990 Symposium on Communication Architectures and Protocols*, pages 200–208, Philadelphia, September 1990.

[55] Steven Ray McCanne. *Scalable Compression and Transmission of Internet Multicast Video*. PhD thesis, University of California at Berkeley, 1996.

[56] David Hutchison, Randa El-Marakby, and Laurent Mathy. A Critique of Modern Internet Protocols: The Issue of Support for Multimedia. In *ECMAST '97: Proceedings of the Second European Conference on Multimedia Applications, Services and Techniques*, pages 507–522, London, UK, 1997. Springer-Verlag.

[57] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: Congestion Control Without Reliability. In *Proc. of the ACM SIGCOMM 2006*, Sep 2006.

[58] Colin Perkins. RTP and the Datagram Congestion Control Protocol (DCCP). Internet-Draft, Work in progress. Available from `http://www.ietf.org/internet-drafts/draft-ietf-dccp-rtp-07.txt`, Jun 2007. expires Dec 2007.

[59] M. Handley, S. Floyd, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448 (Proposed Standard), January 2003.

[60] S. Floyd and E. Kohler. TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant. RFC 4828 (Experimental), April 2007.

[61] Ladan Gharai. RTP with TCP Friendly Rate Control. Internet-Draft, Work in progress. Available from `http://www.ietf.org/internet-drafts/draft-ietf-avt-tfrc-profile-10.txt`, Jun 2007. expires Jan 2008.

[62] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[63] R. Stewart, I. Arias-Rodriguez, K. Poon, A. Caro, and M. Tuexen. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues. RFC 4460 (Informational), April 2006.

[64] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758 (Proposed Standard), May 2004.

[65] Chung-Ming Huang, Ching-Hsien Tsai, and Ming-Chi Tsai. Design and Implementation of Video Streaming Hot-plug between Wired and Wireless Networks Using SCTP. *Oxford Computer Journal*, 49(4):400–417, 2006.

[66] M. Molteni and M. Villari. Using SCTP with Partial Reliability for MPEG-4 Multimedia Streaming. In *Proc. of the 2nd European BSD Conference (BSDCon)*, 2002.

[67] H. Schulzrinne and J. Rosenberg. The Session Initiation Protocol: Internet-Centric Signaling. *IEEE Communications Magazine*, 38(10):134–141, 2000.

[68] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[69] M. Handley, C. Perkins, and E. Whelan. Session Announcement Protocol. RFC 2974 (Experimental), October 2000.

[70] Benjamin W. Wah, Xiao Su, and Dong Lin. A Survey of Error-Concealment Schemes for Real-Time Audio and Video Transmissions over the Internet. In *Proc. of the International Symposium on Multimedia Software Engineering*, pages 17–24, December 2000.

[71] Injong Rhee and Srinath R. Joshi. Error Recovery for Interactive Video Transmission over the Internet. *IEEE Journal on Selected Areas in Communications*, 18(6):1033 – 1049, 2000.

[72] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309 (Informational), April 1998.

[73] Jens-Rainer Ohm. *Multimedia Communication Technology: Representation, Transmission and Identification of Multimedia Signals.* Signals and Communication Technology Engineering. Springer, Berlin, 2004.

[74] Mohammed Al-Mualla, C. Nishan Canagarajah, and David R. Bull. *Video Coding for Mobile Communications : Efficiency, Complexity and Resilience.* Elsevier Academic Press, 2002.

[75] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the Scalable Extension of the H.264/AVC Video Coding Standard. *IEEEă-Trans. on Circuits and Systems for Video Technology*, 17(9):1103–1120, 2007.

[76] Weiping Li. Overview of Fine Granularity Scalability in MPEG-4 Video Standard. *IEEE Trans. on Circuits and Systems for Video Technology*, 11(3):301 – 317, 2001.

[77] Yao Wang and Amy R. Reibman. Multiple Description Coding for Video Delivery. *Proc. of the IEEE*, 93(1):57 – 70, 2005.

[78] M. van der Schaar and H. Radha. Unequal Packet Loss Resilience for Fine-Granular-Scalability Video. *IEEE Trans. on Multimedia*, 3(4):381–393, 2001.

[79] Chien-Peng Ho and Chun-Jen Tsai. Content-Adaptive Packetization and Streaming of Wavelet Video over IP Networks. *EURASIP Journal on Image and Video Processing*, 2007, 2007.

[80] Sergio Daniel Servetto. *Compression and Reliable Transmission of Digital Image and Video Signals.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[81] M. Handley and C. Perkins. Guidelines for Writers of RTP Payload Format Specifications. RFC 2736 (Best Current Practice), December 1999.

[82] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical Loss-Resilient Codes. In *Proc. of the 29th Symposium on Theory of Computing*, pages 150–159, 1997.

[83] A. Albanese, J. Blomer, J. Edmonds, M. Luby, and M. Sudan. Priority Encoding Transmission. *IEEE Trans. on Information Theory*, 42(6):1737 – 1744, 1996.

[84] Hua Cai, Bing Zeng, Guobin Shen, , and Shipeng Li. Error-Resilient Unequal Protection of Fine Granularity Scalable Video Bitstreams. In *Proc. of ICC 2004*, June 20-24 2004.

[85] Amine Bouabdallah and Jérôme Lacan. Dependency-Aware Unequal Erasure Protection Codes. In *Proc. of 15th International Packet Video Workshop, PV2006*, 2006.

[86] J. Kim, R. Mersereau, and Y. Altunbasak. Distributed Video Streaming using Multiple Description Coding and Unequal Error Protection. *IEEE Trans. on Image Processing*, 14(7):849–861, Jul 2005.

[87] Christos Papadopoulos and Guru M. Parulkar. Retransmission-Based Error Control for Continuous Media Applications. *Proceedings of NOSSDAV'96*, 1996.

[88] Lin Ma and Wei Tsang Ooi. Retransmission in distributed media streaming. In *NOSSDAV '05: Proceedings of the international workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 117–122, New York, NY, USA, 2005. ACM Press.

[89] Matthew G. Podolsky, Steven McCanne, and Martin Vetterli. Soft ARQ for Layered Streaming Media. *Journal on VLSI Signal Processing Systems*, 27(1-2):81–97, 2001.

[90] Mei-Hsuan Lu, Peter Steenkiste, and Tsuhan Chen. Time-based Adaptive Retry for Video Streaming in 802.11 WLANs. *Wireless Communications and Mobile Computing, Special Issue on Video Communications for 4G Wireless Systems*, 7(2):187–203, 2007.

[91] C. Wang, R. Chang, J. Ho, and S. Hsu. Rate-Sensitive ARQ for Real-Time Video Streaming. In *Proc. of the Global Telecommunications Conference (GLOBECOM)*, 2003.

[92] Rishi Sinha and Christos Papadopoulos. An Adaptive Multiple Retransmission Technique for Continuous Media Streams. In *NOSSDAV '04*, pages 16–21, Cork, Ireland, 2004.

[93] H. Seferoglu, Y. Altunbasak, O. Gurbuz, and O. Ercetin. Rate Distortion Optimized Joint ARQ-FEC Scheme for Real-Time Wireless Multimedia. In *IEEE International Conference on Communications (ICC 2005)*, pages 1190–1194, May 2005.

[94] F. Zhai, Y. Eisenberg, T. N. Pappas, R. Berry, and A. K. Katsaggelos. Rate-Distortion Optimized Hybrid Error Control for Real-Time Packetized Video Transmission. In *Proc. of the IEEE International Conference on Communications (ICC)*, pages 1318 – 1322, 2004.

[95] A. Majumdar, D.G. Sachs, I.V. Kozintsev, K. Ramchandran, and M.M. Yeung. Multicast and unicast real-time video streaming over wireless LANs. *IEEE Trans. on Circuits and Systems for Video Technology*, 12(6):524–534, 2002.

[96] C. Q. Yang, E. Hossain, and V. K. Bhargava. On Adaptive Hybrid Error Control in Wireless Networks using Reed-Solomon Codes. *IEEE Trans. on Wireless Communications*, 4(3):835 – 840, 2005.

[97] Abdelhamid Nafaa and Ahmed Mehaoua. Joint Loss Pattern Characterization and Unequal Interleaved FEC Protection for Robust H.264 Video Distribution over Wireless LAN. *Computer Networks*, 49(6):766–786, 2005.

[98] A. Nafaa, T. Ahmed, and A. Mehaoua. Unequal and Interleaved FEC Protocol for Robust MPEG-4 Multicasting over Wireless LANs. In *IEEE International Conference on Communications*, pages 1431 – 1435, 2004.

[99] Huahui Wu, Mark Claypool, and Robert Kinicki. Adjusting Forward Error Correction with Quality Scaling for Streaming MPEG. In *Proc. of NOSSDAV'05*, June 2005.

[100] S. H. Kang and A. Zakhor. Packet Scheduling Algorithm for Wireless Video Streaming. In *Proc. of the International Packet Video Workshop*, 2002.

[101] P. Pahalawatta, R. Berry, T. Pappas, and A. Katsaggelos. Content-Aware Resource Allocation and Packet Scheduling for Video Transmission over Wireless Networks. *IEEE Journal on Selected Areas in Communications*, 25(4):749–759, 2007.

[102] Srivatsan Varadarajan, Hung Q. Ngo, and Jaideep Srivastava. Error Spreading: A Perception-Driven Approach to Handling Error in Continuous Media Streaming. *IEEE/ACM Trans. on Networking*, 10(1):139–152, 2002.

[103] Jeong-Yong Choi and Jitae Shin. A Novel Content-Aware Interleaving for Wireless Video Transmission. *Computer Communications*, 29(13-14):2634–2645, 2006.

[104] Charles Krasic and Jonathan Walpole. Quality-adaptive Media Streaming by Priority Drop. In *Proc. of the 13th NOSSDAV Workshop*, pages 112–121, 2003.

[105] Wei Tu, Jacob Chakareski, and Eckehard Steinbach. Rate-Distortion Optimized Frame Dropping and Scheduling for Multi-User Conversational and Streaming Video. In *Proc. of 15th International Packet Video Workshop (PV2006)*, pages 864–872, Apr 2006.

[106] Damir Isovic and Gerhard Fohler. Quality Aware MPEG-2 Stream Adaptation in Resource Constrained Systems. In *Proc. of 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, pages 23–32, Catania, Italy, 30 June – 2 July 2004.

[107] A.K. Katsaggelos, Y. Eisenberg, F. Zhai, R. Berry, and T.N. Pappas. Advances in Efficient Resource Allocation for Packet-Based Real-Time Video Transmission. *Proc. of the IEEE*, 93(1):135 – 147, 2005.

[108] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[109] Lefteris Mamatas, Tobias Harks, and Vassilis Tsaoussidis. Approaches to Congestion Control in Packet Networks. *Journal of Internet Engineering (JIE)*, 1(1), 2007.

[110] Nguyen Dieu Thanh and Ostermann Joern. Streaming and Congestion Control using Scalable Video Coding based on H.264/AVC. *Journal of Zhejiang University - Science A*, 7(5):749–754, 2006.

[111] C. Huang and L. Xu. SRC: Stable Rate Control for Streaming Media. In *Proc. of GLOBECOM*, pages 4016–4021, 2003.

[112] Shanwei Cen, Jonathan Walpole, and Calton Pu. Flow and Congestion Control for Internet Media Streaming Applications. In *Proc. SPIE Multimedia Computing and Networking*, pages 250–264, 1998.

[113] Sally Floyd, Mark Handley, Jitendra Padhye, and Jorg Widmer. Equation-based Congestion Control for Unicast Applications. In *SIGCOMM 2000*, pages 43–56, Stockholm, Sweden, August 2000.

[114] R. Rejaie, D. M. Handley, and D. Estrin. Layered Quality Adaptation for Internet Video Streaming. *IEEE Journal on Selected Areas in Communications*, 18(12):2530–2543, 2000.

[115] Nikolaos Laoutaris and Ioannis Stavrakakis. Intrastream Synchronization for Continuous Media Streams: A Survey of Playout Schedulers. *IEEE Network Magazine*, 16(3), May 2002.

[116] Ernst Biersack and Werner Geyer. Synchronized Delivery and Playout of Distributed Stored Multimedia Streams. *Multimedia Systems*, 7(1):70–90, 1999.

[117] Marco Roccetti, Vittorio Ghini, Giovanni Pau, Paola Salomoni, and Maria Elena Bonfigli. Design and Experimental Evaluation of an Adaptive Playout Delay Control Mechanism for Packetized Audio for Use over the Internet. *Multimedia Tools and Applications*, 14(1):23–53, 2001.

[118] Azzedine Boukerche, Sungbum Hong, and Tom Jacob. An Efficient Synchronization Scheme of Multimedia Streams in Wireless and

Mobile Systems. *IEEE Trans. on Parallel and Distributed Systems*, 13(9):911–923, 2002.

[119] M. Kalman, E. Steinbach, and B. Girod. Adaptive Media Playout for Low Delay Video Streaming over Error-Prone Channels. *IEEE Trans. on Circuits and Systems for Video Technology*, 14(6):841–851, Jun 2004.

[120] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. on Computer Systems*, 18(3):263–297, 2000.

[121] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proc. of OSDI'04*, San Francisco, CA, December 2004.

[122] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[123] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Conference on Innovative Data Systems Research, CIDR*, 2003.

[124] A.E. Ozcan, O. Layaida, and J.-B. Stefani. A component-based approach for MPSoC SW design: experience with OS customization for H.264 decoder. In *3rd Workshop on Embedded Systems for Real-Time Multimedia*, 2005.

[125] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[126] M. Duller, R. Tamosevicius, G. Alonso, and D. Kossmann. XTream: Personal Data Streams. In *Proc. of SIGMOD 2007*, 2007.

[127] Emmanuel Bouix, Philippe Roose, Marc Dalmau, and Franck Luthon. A Component Model for Transmission and Processing

of Synchronized Multimedia Data Flows. In *Proc. of the First International Conference on Distributed Frameworks for Multimedia Applications*, pages 45–53, Washington, DC, USA, 2005.

[128] Baochun Li and K. Nahrstedt. A Control-Based Middleware Framework for Quality of Service Adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632–1650, 1999.

[129] Alexander Eichhorn and Winfried E. Kühnhauser. A Component-based Architecture for Streaming Media. In *Proc. of the Net.Objectdays*, pages 273–286, September 2001.

[130] G. Coulson, S. Baichoo, and O. Moonian. A Retrospective on the Design of the GOPI Middleware Platform. *ACM Multimedia Journal*, 8(5):340–352, 2002.

[131] Frank Siqueira and Vinny Cahill. Quartz: A QoS Architecture for Open Systems. In *International Conference on Distributed Computing Systems*, pages 197–204, 2000.

[132] Thomas Plagemann et al. Flexible and Extensible QoS Management for Adaptable Middleware. In *Proc. of International Workshop on Protocols for Multimedia Systems (PROMS 2000)*, Cracow, Poland, Oct 2000.

[133] D. Donaldson et al. Dimma – A Multimedia ORB. In *Proc. of the ACM Middleware 98*, pages 141–156, 1998.

[134] International Standards Organization. Information Technology; Open Distributed Processing, 1998.

[135] Yuqing Song Aidong Zhang and Markus Mielke. NetMedia: Streaming Multimedia Presentations in Distributed Environments. *IEEE Multimedia Magazin*, 9(1):56–73, 2002.

[136] Cyrus Shahabi, Roger Zimmermann, Kun Fu, Shu-Yuen, and Didi Ya. Yima: A Second-Generation Continuous Media Server. *IEEE Computer*, 35:56 – 64, Jun 2002.

[137] Duangdao Wichadakul. *Q-Compiler: Meta-Data QoS-Aware Programming and Compilation Framework*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[138] Xiaohui Gu, Klara Nahrstedt, and Bin Yu. SpiderNet: An Integrated Peer-to-Peer Service Composition Framework. In *Proc. of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 110–119, Washington, DC, USA, 2004.

[139] Klara Nahrstedt, Bin Yu, Jin Liang, and Yi Cui. Hourglass Multimedia Content and Service Composition Framework for Smart Room Environments. *Pervasive Mob. Comput.*, 1(1):43–75, 2005.

[140] Asfandyar Qureshi and John Guttag. Horde: Separating Network Striping Policy from Mechanism. In *3rd International Conference on Mobile Systems, Applications, and Services (Mobisys 2005)*, Seattle, WA, June 2005.

[141] Franz Kalleitner, Mario Konegger, Attila Takćs, and Ákos Kovács. M-Pipe - A novel Media Delivery Framework. In *Proc. of the European Symposium on Mobile Media Delivery*, 2006.

[142] A. Takacs, A. Kovacs, F. Kalleitner, and H. Brand. Forward Information - A General Approach for Scalable audiovisual Service Delivery. In *2nd International Symposium on Wireless Communication Systems*, pages 158– 162, 2005.

[143] Chih-Ming Chen, Chia-Wen Lin, and Yung-Chang Chen. Packet Scheduling for Video Streaming over Wireless with Content-Aware Packet Retry Limit. In *Workshop on Multimedia Signal Processing*, pages 409–414, 2006.

[144] Jitae Shin, JongWon Kim, and C.-C. Jay Kuo. Quality of Service Mapping Mechanism for Packet Video in Differentiated Services Network. *IEEE Trans. on Multimedia*, 3(2):219–231, June 2001.

[145] Vincent Lecuire, Francis Lepage, and Khalil Kammoun. Enhancing Quality of MPEG Video through Partially Reliable Transport Service in Interactive Application. In *MMNS '01: Proceedings of the 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, pages 96–109, London, UK, 2001.

[146] Umar Choudhry and JongWon Kim. Performance Evaluation of H.264 Mapping Strategies over IEEE 802.11e WLAN for Robust Video Streaming. In *Pacific-Rim Conf. on Multimedia*, pages 818–829, 2005.

[147] Damir Isović. *Flexible Scheduling for Media Processing in Resource Constrained Real-Time Systems.* PhD thesis, Mälardalen University, 2004.

[148] J. Chakareski, J. Apostolopoulos, S. Wee, W. Tan, and B. Girod. Rate-Distortion Hint Tracks for Adaptive Video Streaming. *IEEE Trans. on Circuits and Systems for Video Technology*, 15(10):1257–1269, Oct 2005.

[149] Y. Liang, J. Apostolopoulos, and B. Girod. Analysis of packet loss for compressed video: Does burst-length matter. In *Proc. of IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, 2003.

[150] Y.J. Liang and B. Girod. Network-adaptive low-latency Video Communication over Best-effort Networks. *IEEE Trans. on Circuits and Systems for Video Technology*, 16(1):72–81, 2006.

[151] M. Kalman and B. Girod. Techniques for Improved Rate-Distortion Optimized Video Streaming. *Stanford Journal of Research*, 2(1):45–54, Nov 2005.

[152] Jacob Chakareski and Pascal Frossard. Rate-Distortion Optimized Packet Scheduling over Bottleneck Links. In *Proc. of the IEEE ICME*, Jul 2005.

[153] M. Röder, J. Cardinal, and R. Hamzaoui. Efficient Rate-Distortion Optimized Media Streaming for Tree-Reducible Packet Dependencies. In *Proc. of Multimedia Computing and Networking*, 2006.

[154] Gene Cheung and Wai-Tian Tan. Directed Acyclic Graph based Source Modeling for Data Unit Selection of Streaming Media over QoS Netwo rks. In *Int. Conf. Multimedia & Expo*, 2002.

[155] Martin Röder, Jean Cardinal, and Raouf Hamzaoui. Branch and bound Algorithms for Rate-distortion optimized Media Streaming. *IEEE Trans. on Multimedia*, 8(1):170–178, 2006.

[156] Franz Kalleitner, Mario Konegger, Attila Takćs, and Ákos Kovács. A Cross-Layer Framework for Content based fine-grained Scheduling of Audiovisual Streams over Wireless Network. In *Proc. of the IASTED CIIT*, 2005.

[157] L.-A. Larzon, U. Bodin, and O. Schelen. Hints and Notifications. In *IEEE Wireless Communications and Networking Conference*, pages 635–641, 2002.

[158] D. Clark J. Saltzer, D. Reed. End-to-end Arguments in System Design. *ACM Trans. on Computer Systems*, 2(4):277–288, 1984.

[159] Jiantao Kong and Karsten Schwan. KStreams: Kernel Support for Efficient Data Streaming in Proxy Servers. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 159–164, New York, NY, USA, 2005.

[160] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[161] Y. Eisenberg, F. Zhai, T.N. Pappas, R. Berry, and A.K. Katsaggelos. VAPOR: Variance-Aware per-pixel Optimal Resource Allocation. *IEEE Trans. on Image Processing*, 15(2):289–299, 2006.

[162] Enrico Masala and Juan Carlos De Martin. Analysis-by-Synthesis Distortion Computation for Rate-Distortion Optimized Multimedia Streaming. In *ICME Multimedia and Expo*, 2003.

[163] Alexander Eichhorn and Winfried E. Kühnhauser. Datenströme in multimedialen Systemen. In *Proc. of Net.Objectdays – 8th Workshop on Multimedia Informations and Communication Systems*, pages 687–695, October 2002.

[164] Gary Sullivan and Thomas Wiegand. Video Compression - From Concepts to the H.264/AVC Standard. *Proc. of the IEEE, Special Issue on Advances in Video Coding and Delivery*, 93(1):18–31, Jan 2005.

[165] J. Chakareski, S. Han, and B. Girod. Layered Coding vs. Multiple Descriptions for Video Streaming over Multiple Paths. In *Proc. of ACM Multimedia*, pages 422–431, 2003.

[166] Martin Hoffmann and Winfried E. Kühnhauser. Towards a Structure-Aware Failure Semantics for Streaming Media Communication Models. *Journal of Parallel and Distributed Computing*, 65(9):1047–1056, September 2005.

[167] Matthew Luckie, Kenjiro Cho, and Bill Owens. Inferring and Debugging Path MTU Discovery Failures. In *Proc. of the Internet Measurement Conference 2005 on Internet Measurement Conference*, pages 17–17, Berkeley, CA, USA, 2005.

[168] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Trans. on Image Processing*, 13(4):600–612, 2004.

[169] G. E. P. Box and D. R. Cox. An Analysis of Transformations. *Journal of the Royal Statistical Society*, pages 211–243, discussion 244–252, May 1964.

[170] Z. Wang, L. Lu, and A. Bovik. Video Quality Assessment based on Structural Distortion Measurement. *Signal Processing: Image Communication, special issue on objective video quality metrics*, 19(2):121–132, 2004.

[171] E. N. Gilbert. Capacity of a burst-noise channel. *Bell Systems Technical Journal*, 39:1253–1266, Sept 1960.

[172] Stephan Wenger et al. Codec Control Messages in the RTP Audio-Visual Profile with Feedback (AVPF). Internet-Draft, Work in progress. Available from `http://www.ietf.org/internet-drafts/draft-ietf-avt-avpf-ccm-10.txt`, May 2007. expires Apr 2008.

[173] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, 2(1):39–59, Feb 1984.

[174] P. Narasimban et al. Using Interceptors to Enhance CORBA. *IEEE Computer*, 32(7):62 – 68, 1999.

[175] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210 (Proposed Standard), September 1997.

[176] S. Shenker, C. Partridge, and R. Guerin. Specification of Guaranteed Quality of Service. RFC 2212 (Proposed Standard), September 1997.

[177] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented Software Architecture Vol 2: Patterns for Concurrent and Networked Objects*. John Wiley and Sons Ltd, 2000.

[178] M. Welsh, S. Gribble, E. Brewer, and D. Culler. A design framework for highly concurrent systems. Technical Report UCB/CSD-00-1108, UC Berkeley CS, October 2000.

[179] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Trans. on Computer Systems*, 18(1):37–66, Feb 2000.

[180] T.-W. A. Lee, S.-H. G. Chan, Q. Zhang, W.-W. Zhu, and Y.-Q. Zhang. Optimal Allocation of Packet-Level and Byte-level FEC in Video Multicasting over Wired and Wireless Networks. In *Proc. of GLOBECOM*, pages 1994–1998, 2001.

[181] Y. Wang, W. Huang, and J. Korhonen. A Framework for Robust and Scalable Audio Streaming. In *Proc. of ACM Multimedia 2004*, New York, NY, USA, October 2004.

[182] Nick Feamster, Deepak Bansal, and Hari Balakrishnan. On the Interactions Between Layered Quality Adaptation and Congestion Control for Streaming Video. In *11th International Packet Video Workshop*, Kyongju, Korea, April 2001.

[183] Z. Fu, X. Meng, and S. Lu. A Transport Protocol for Supporting Multimedia Streaming in Mobile Ad-hoc Networks. *IEEE Journal on Selected Areas in Communications*, 21(10):1615–1626, 2004.

[184] Cormac J. Sreenan, Jyh-Cheng Chen, Prathima Agrawal, and B. Narendran. Delay Reduction Techniques for Playout Buffering. *IEEE Trans. on Multimedia*, 2(2):88–100, 2000.

[185] Christian Brien. Synchronisation in einem stromorientierten Kommunikationsmodell. Diploma thesis, Technische Universität Ilmenau, 2005.

[186] Mario Holbe. Stromorientierte ereignisbasierte Kommunikation unter Echtzeitbedingungen. Diploma thesis, Technische Universität Ilmenau, 2004.

[187] Stephan Wenger, Ye kui Wang, and Miska M. Hannuksela. RTP Payload Format for H.264/SVC Scalable Video Coding. *Journal of Zhejiang University - Science A*, 7(5):657–667, 2006.