

Universität des Saarlandes  
Naturwissenschaftlich-Technische Fakultät I  
Fachrichtung Informatik  
Lehrstuhl für Computergraphik

Bachelor-Arbeit

# ”Integration von unterschiedlichen Methoden zur Fehlerkorrektur in NMM”

Jan Ruffing

2. September 2009

Unter der Leitung von  
Prof. Dr. Philipp Slusallek

Betreut von  
Michael Replinger

Begutachtet von  
Prof. Dr. Philipp Slusallek  
Prof. Joachim Weickert

Ich danke ...

... Prof. Slussalek für die Vergabe dieses Bachelor-Themas.

... Michael Repplinger für die hervorragende Betreuung.

... meiner Familie.

## **Eidesstattliche Erklärung**

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

Saarbrücken, 2. September 2009

---

Jan Ruffing

## **Einverständniserklärung**

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

Saarbrücken, 2. September 2009

---

Jan Ruffing

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.1.1	Reduzierte Bandbreite . . . . .	1
1.1.2	Verlustbehafteter Datentransfer . . . . .	2
1.2	Ziele . . . . .	2
1.3	Aufbau der Arbeit . . . . .	2
<b>2</b>	<b>Grundlagen</b>	<b>4</b>
2.1	Einführung in die Vorwärtsfehlerkorrektur . . . . .	4
2.1.1	Fehler . . . . .	4
2.1.2	Vorwärtsfehlerkorrektur . . . . .	5
2.1.3	Parameter von FECs . . . . .	5
2.1.4	Abschätzung der Korrekturrate nach oben . . . . .	6
2.2	Fehler-Szenarien . . . . .	6
2.2.1	Rauschen . . . . .	7
2.2.2	Blockfehler . . . . .	7
2.2.3	Zwischenstufen . . . . .	7
2.3	Existierende FEC Codes . . . . .	7
2.3.1	Viterbi Code . . . . .	9
2.3.2	Reed-Solomon Code . . . . .	10
2.3.3	Turbo Code . . . . .	11
2.3.4	Low-Density-Parity-Check Code . . . . .	11
2.4	Codec-Auswahl . . . . .	13
2.4.1	Entscheidung für LDPC . . . . .	13
2.4.2	LDPC FEC Codec von Planète-BCAST . . . . .	14
<b>3</b>	<b>Netzwerk-Integrierte Multimedia-Middleware</b>	<b>16</b>
3.1	Flussgraph . . . . .	16
3.1.1	Knoten . . . . .	17
3.1.2	Jacks . . . . .	17
3.2	Kommunikation . . . . .	17
3.2.1	Nachrichten . . . . .	17
3.3	Transportstrategien . . . . .	19
3.3.1	Serialisierungs-Streams . . . . .	19
3.3.2	Netzwerk-Streams . . . . .	19
<b>4</b>	<b>Integration von FEC in NMM</b>	<b>20</b>
4.1	Anwendungsszenario . . . . .	20

## Inhaltsverzeichnis

4.2	Aufbau einer FECTransportStrategy . . . . .	21
4.3	Test-Streams . . . . .	21
<b>5</b>	<b>Implementierung</b>	<b>23</b>
5.1	FECStream . . . . .	24
5.1.1	DummyFECStream . . . . .	25
5.1.2	LDPCFECStream . . . . .	25
5.2	Support Streams . . . . .	25
5.2.1	BufferedOutputStream . . . . .	25
5.2.2	LossToleratingPacketStream . . . . .	26
5.2.3	UnbufferedNetStream . . . . .	26
5.3	Error Streams for Testing . . . . .	26
5.3.1	BitErrorStream . . . . .	26
5.3.2	PacketLossErrorStream . . . . .	26
5.4	fec.test Testanwendung . . . . .	26
<b>6</b>	<b>Testen der Implementierung</b>	<b>28</b>
6.1	Testumgebung . . . . .	28
6.2	Fehlersimulation mittels BlockErrorStream . . . . .	28
6.2.1	Starten des Testprogramms . . . . .	29
6.2.2	Ergebnis . . . . .	30
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>32</b>
7.0.3	Erreichte Ziele . . . . .	32
7.0.4	Erweiterungsmöglichkeiten . . . . .	32
	<b>Quellen- und Literaturverzeichnis</b>	<b>34</b>
<b>A</b>	<b>Quellcode</b>	<b>36</b>

# Abbildungsverzeichnis

2.1	Schematische Darstellung des Datentransfers . . . . .	5
2.2	Messung der Korrektur-Rate eines Viterbi-Codes mit Datenrate 1/2 (aus [3] . .	9
2.3	Schematische Darstellung des Viterbi-Algorithmus . . . . .	10
2.4	Der Reed-Solomon-Algorithmus' kann FEC-Zeichen/2 Fehler sicher ausgleichen	10
2.5	Schematische Darstellung des Turbo Code-Prinzips . . . . .	11
2.6	Korrektur-Rate eines Turbo-Codes (aus [13]) . . . . .	12
2.7	Korrektur-Rate eines kommerziellen LDPC-Codes in Satelitten-Modems (aus [7])	13
2.8	Korrektur-Rate von Viterbi, Viterbi-ReedSolomon und Turbo-Code, aus [19] .	14
3.1	Darstellung eines Flussgraphen zum Abspielen von mp3s . . . . .	16
3.2	Darstellung der Verbindung zweier Jacks . . . . .	18
3.3	Serialisierung: Ein NMM-Objekt wird in mit Indizes versehene Binärdaten um- gewandelt, welche dann ans Netzwerk weitergereicht werden . . . . .	19
4.1	Einbau von FEC in die Transportstrategie: Das Encodieren findet zwischen dem Buffern der serialisierten Daten und dem Senden ans Netzwerk statt . . . . .	21
5.1	Ableitungs-Hierarchie der ausgehenden Streams . . . . .	23
5.2	Ableitungs-Hierarchie der eingehenden Streams . . . . .	23
5.3	Parameter von fec_test . . . . .	27
6.1	Transportstrategien von Client und Server . . . . .	28

# Tabellenverzeichnis

2.1	Tabelle mit maximal erreichbaren Datenraten nach Shannon . . . . .	6
2.2	Umrechnungstabelle $E_b/N_o$ in Fehlerrate . . . . .	7
6.1	Messergebnis für LDPC, Fehlersimulation mittels BlockErrorOStream . . . . .	31

# Kapitel 1

## Einleitung

### 1.1 Motivation

In den letzten Jahren sind drahtlose Lösungen für den Datentransfer zwischen Computern immer beliebter geworden: Drahtlose Netzwerke (*wireless local area networks, WLANs*) sind mittlerweile verbreitet, und bieten aufgrund der Kabellosigkeit einen deutlichen Gewinn an Komfort und Mobilität. Gleichzeitig hat die Datenmenge, die drahtlos übertragen wird, immer weiter zugenommen. Dies liegt zum einen in der Natur des Mediums: Da sich mehrere Rechner die Bandbreite einer zentrale Sendestation (*WLAN-Router*) teilen, nimmt mit stärkerer Nutzung die individuelle Bandbreite ab. Zum anderen werden Computer immer stärker als Multimedia-Geräte genutzt, wobei die Daten zunehmend live über das Netzwerk zur Verfügung gestellt werden. Gerade beim Abspielen auf mobilen Endgeräten wie Laptops bietet sich hier eine drahtlose Lösung an.

Trotzdem hat der Datentransfer über WLAN zwei entscheidende Nachteile:

#### 1.1.1 Reduzierte Bandbreite

Multimedia-Daten verbrauchen oft viel Bandbreite (HDTV-Datenströme benötigen zwischen 22 und 27 MBit/s, während WLAN in der Praxis eine Bandbreite um 25 MBit/s zur Verfügung stellt). Diese Daten sind ausserdem oft zeit-kritisch; Verzögerungen beim Übertragen der Daten können das Multimedia-Erlebnis beeinträchtigen. Entweder wird das Live-Erlebnis pausiert, um weitere Daten zwischenspeichern (*buffern*). Oder es kommt zu Sprüngen und Artefakten in Ton und Bild wenn, die Daten aufgrund der Überlastung des Netzwerks nicht rechtzeitig zum Abspielen übertragen worden sind. Hier kann es erforderlich sein, die Datenrate zu reduzieren. Bei Multimediadaten beispielsweise durch das Reduzieren der Framerate oder der Auflösung. Eine solche Anpassung der Multimediadaten lässt sich mit der *Network-Integrated Multimedia Middleware (NMM)* bereits umsetzen, einem Framework, das den verteilten Bearbeiten von Multimedia Daten ermöglicht.

## 1.1.2 Verlustbehafteter Datentransfer

Ein weiteres Problem ist, dass drahtlose Lösungen deutlich unzuverlässiger sind als leitungs-basierte Lösungen. Selbst wenn die Datenrate der Verbindung nicht ausgeschöpft wird können Datenpakete während der Übertragung verloren gehen, etwa durch Interferenzen mit andere sendende Geräte (WLAN-Netze, Mikrowellen, Bluetooth), Störungen durch Hindernisse wie Wände oder Störungen aufgrund von Bewegung oder Abstand der Sender.

Diese Probleme - Zunahme des zeitkritischen Datenaufkommens über unzuverlässige Verbindungen - machen auch vor der *Network-Integrated Multimedia Middleware (NMM)* nicht halt. Derzeit bietet NMM keine Möglichkeit zum sicheren Datentransfer. Zuverlässige Transferprotokolle (wie etwa TCP) werden nicht verwendet. Diese bieten gegenüber UDP auch nur eine deutlich reduzierte Bandbreite.

Hier bieten sich Methoden zur Forwärtsfehlerkorrektur (*Forward Error Correction, FEC*) an: Es werden zusätzlich redundante Informationen übertragen, welche es ermöglichen durch Störungen verlorengegangene Informatuinen bis zu einem gewissen Grad auszugleichen: Datenverluste werden reduziert oder ganz verhindert.

## 1.2 Ziele

Das Ziel dieser Arbeit ist es, NMM um solche FEC-Mechaniken zu erweitern. Hierbei sollen verschiedene Szenarien berücksichtigt werden, etwa mit einer unterschiedlichen Gewichtung von Datenrate und Fehlerkorrektur. Um NMM nicht auf ein Szenario festzulegen ist ein allgemeines Framework notwendig, in das dann prinzipiell alle FECs integriert werden können.

Entsprechend ergeben sich im Rahmen dieser Arbeit folgende Zielsetzungen

- Es sollen existierende FEC-Methoden untersucht werden. Hier geht es darum, Unterschiede zwischen den FECs, und ihre Eignung für bestimmte Störungsszenarie festzustellen. Geeignete Codecs sollen für die Integration in NMM ausgewählt werden.
- Die Netzwerk-Ebene von NMM soll um ein entsprechendes Framework erweitert werden, so dass sie diese Codecs verwenden kann.
- Außerdem soll eine Testumgebung geschaffen werden, die das Überprüfen der Leistungsfähigkeit der Codecs in NMM ermöglicht.

## 1.3 Aufbau der Arbeit

Das zweite Kapitel wird das grundlegenden Konzept der Forwärtsfehlerkorrektur erklärt, und existierende FEC Methoden vorgestellt. Hier werden auch die Codecs präsentiert, die für die Auswahl in NMM ausgewählt wurden.

Das dritte Kapitel wird die Network-Integrated Multimedia Middleware vorstellen - die grundlegende Architektur. Hier wird insbesondere auf Aspekte der Datenübertragung weiter eingegangen, die für das Thema dieser Arbeit von besonderem Interesse sind.

## *Kapitel 1 Einleitung*

Das vierte Kapitel wird sich mit der Integration von FEC-Codecs in NMM befassen.

Das fünfte Kapitel zeigt dann die Umsetzung dieses Konzept anhand der prototypischen Einbindung eines FEC-Codecs.

Das sechste Kapitel wird sich mit der Auswertung dieses Prototyps befassen. Dazu wird die Messumgebung vorgestellt, ebenso wie sie durchgeführten Tests und deren Ergebnisse.

Im siebten Kapitel werden die Ergebnisse nochmal zusammengefaßt und ein Ausblick auf mögliche Erweiterungen wird gewährt.

# Kapitel 2

## Grundlagen

### 2.1 Einführung in die Vorwärtsfehlerkorrektur

#### 2.1.1 Fehler

Die zwischen *Sender* und *Empfänger* übertragenen Daten werden als *Signal* bezeichnet. Beim Übertragen des Signals über eine unsichere Leitung kann es durch Störungen dazu kommen, dass Informationen verändert werden oder verloren gehen. Das Verändern einer Information wird allgemein als *Fehler* bezeichnet. Hierbei läßt sich differenzieren, je nachdem in welche Einheit man die Daten einteilt:

- Ein *Bit-Fehler* betrifft genau ein Bit.
- Betrachtet man als Informations-Einheit ein *Zeichen* von mehreren Bit Länge (etwa ein Byte), so wird jede Veränderung am Zeichen als *Zeichen-Fehler* bezeichnet, unabhängig ob ein oder mehrere Bits abweichen.
- Ab der Transportprotokoll-Ebene hat man es hingegen meist mit dem Verlust von Transport-Paketen von meist mehreren Kilobyte Länge zu tun. Hier spricht man dann von *Paket-Fehlern*.

Unter der *Fehlerrate* versteht man jetzt das Verhältnis von fehlerhaften Einheiten zu den insgesamt gesendeten Einheiten. Je nachdem, welche Einheit betrachtet wird, kann es hier zu Abweichungen kommen. Beispiel: Vier Bitfehler in einem 32bit-Signal entsprechen einer Bit-Fehlerrate von 12,5%. Je nachdem, in welche Bytes sie auftreten könnten sie aber einer Byte-Fehlerrate zwischen 25% und 100% entsprechen. Das Verhältnis von korrekt übertragenen Einheiten zu fehlerhaft übertragenen Einheiten wird *Störabstand* (*Signal-to-Noise-Ratio*, *SNR*) genannt; dieser wird oft im logarithmischen Maßstab dargestellt. Der normalisierte Störabstand wird häufig mit  $E_b/N_o$  bezeichnet, und dient als grobe Vergleichsmöglichkeit unabhängig von Einheit und Bandbreite.

## 2.1.2 Vorwärtsfehlerkorrektur

Vorwärtsfehlerkorrektur (*Forward Error Correction, FEC*) ist eine Methode, die dazu dient die *Fehlerrate* zu senken, oder wenn möglich ganz auf Null zu reduzieren. Hierzu werden die *Nutzdaten encodiert*: Es werden aus den Nutzdaten redundante Daten (*FEC-Daten, Error-Code*) errechnet. Beim Encodieren der Daten wird aus einer Sequenz von  $m$  Zeichen (meist Bytes) mittels eines Algorithmus ein *Signalwort* (auch *Code-Wort*) von  $m+n$  Zeichen berechnet. Dies erlaubt es dem Empfänger, beim *Dekodieren* Fehler im *Signal* aus Daten und redundanten Daten zu erkennen und zu korrigieren.

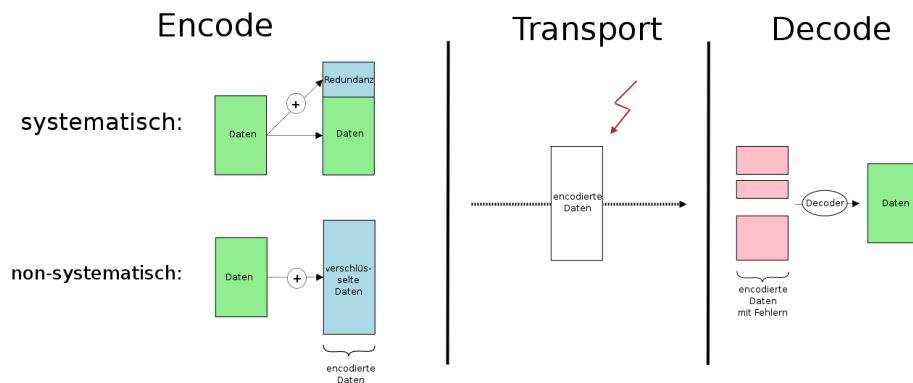


Abbildung 2.1: Schematische Darstellung des Datentransfers

Hierbei lassen sich verschiedene Fälle unterscheiden: Beim *systematischen Enkodieren* wird der Error-Code zusätzlich zu den Daten generiert, während beim *non-systematischen Enkodieren* im Signal die Daten und der Error-Code nicht mehr klar zu trennen sind. Die meisten FEC-Algorithmen sind systematisch, und wenn im folgenden Text von Enkodieren gesprochen wird, kann von systematischem Enkodieren ausgegangen werden.

Eine weitere Unterscheidungsmöglichkeit besteht darin, auf welche Weise die zu encodierenden Daten an den Encoder gegeben werden. Bei *Block-Codes* wird aus einem Daten-Block (auch *Buffer*) mit vorgegebener Größe ein Signal-Block erzeugt. Bei *Faltungscodes* "wandert" der Encodierer hingegen über die Eingangsdaten, was das kontinuierliche Daten- und Signalströme ermöglicht. Durch die Eingabe von Daten-Blöcken ist hier aber auch eine Handhabung analog zu Block-Codes möglich. Die Möglichkeit, mit festen Buffer-Größen zu Arbeiten ist für die Integration in NMM von Interesse.

## 2.1.3 Parameter von FECs

Bei beiden Verfahren stehen die Daten-Länge und die Signal-Länge in einem fixen Verhältnis, der *Datenrate*. Je geringer die Datenrate, desto mehr Bandbreite wird zusätzlich benötigt.

Entscheidend zur Wiederherstellung beschädigter Zeichen ist die *Hamming-Distanz* - die bitweise Abweichung zweier (Signal-)Wörter. Sie bestimmt die Möglichkeiten zur Fehlererkennung

und zur Fehlerkorrektur. Bei einer Hamming-Distanz  $h$  können  $h-1$  fehlerhafte Zeichen im Wort erkannt werden, und es können  $(h-1)/2$  fehlerhafte Zeichen korrigiert werden (abgerundet).

Als *Korrekturrate* bezeichnen wir die Zahl der korrigierbaren Zeichen je Signallänge. Dies ist ein statistischer Wert, da die Zahl der auffindbaren Fehler je nach Wortkombination und Position unterschiedlich sein können. Deswegen unterscheiden wir zusätzlich noch die *garantierte Korrekturrate* die angibt, wieviele Fehler auf jeden Fall korrigiert werden können.

Hier gibt es zwischen den verschiedenen Codes gravierende Unterschiede, und auch innerhalb einer Code-Klasse gibt es Abstufungen. Innerhalb einer Code-Klasse kann anhand folgender Parameter differenziert werden:

- Zahl der beim Encodieren betrachteten Zeichen
- Zahl der beim Encodieren erzeugten Code-Zeichen
- Parameter, die die Komplexität der Berechnung festlegen

In der Praxis schlägt sich das dann in benötigter Rechenleistung und benötigter Bandbreite einerseits, und der Korrekturrate und garantierten Korrekturrate andererseits nieder.

### 2.1.4 Abschätzung der Korrekturrate nach oben

Für einen Übertragungskanal, dessen Fehlerschema dem additiven weißen gaußschen Rauschen entspricht (gleichmäßig verteilte Fehler, wobei 0en und 1en gleich fehleranfällig sind), hat Shannon in [12] folgende theoretische Obergrenze hergeleitet:

$$\text{Maximale Datenrate} = \text{Bandbreite} \cdot \log_2(1 + \text{Störabstand})$$

Die Bandbreite wird dabei in Hz angegeben wird; WLAN nach Standard IEEE 802.11g bietet eine Brutto-Bandbreite bis zu 40 MHz. Die maximale Datenrate gibt die maximal fehlerfrei transportierbaren Nutzdaten an wird in bit/s angegeben.

Fehlerrate	SNR	Maximale Datenrate
33%	2	$1,37 \cdot B$
25%	3	$1,58 \cdot B$
20%	4	$1,81 \cdot B$
10%	9	$3,16 \cdot B$
5%	19	$6,33 \cdot B$

Tabelle 2.1: Tabelle mit maximal erreichbaren Datenraten nach Shannon

## 2.2 Fehler-Szenarien

Bei der Übertragung von Daten über WLAN gibt es viele Fehlerursachen. Die resultierenden Fehler lassen sich grob in zwei Schemata einteilen:

### 2.2.1 Rauschen

*Rauschen (Noise)* bezeichnet eine über einen längeren Zeitraum gleichmäßige Fehlerwahrscheinlichkeit. Interferenzen durch andere Geräte wie andere WLAN-Netze oder Mikrowellen, aber auch statische Hindernisse wie Wände führen meist zu diesem Fehler-Schema.

### 2.2.2 Blockfehler

*Blockfehler (Burst-Fehler)* bezeichnen das oft plötzliche Auftreten mehrerer Fehler in Folge, meist über einen begrenzten Zeitraum. Typische Fehlerquellen wären etwa eine *Überlastung* der Verbindung (*Many-to-Many*): Die Senderate überschreitet die Bandbreite der Verbindung, oder auf Empfängerseite können die Daten nicht schnell genug verarbeitet werden, so dass mehrere Datenpakete in Folge verloren gehen. Auch *Synchronisationsfehler* als Folge einer kurzen Verbindungsstörung führen meist zu diesem Fehler-Schema.

### 2.2.3 Zwischenstufen

Es sind auch Zwischenstufen denkbar, etwa wenn die Fehlerwahrscheinlichkeit sich über einen kurzen Zeitraum deutlich erhöht. Ursachen könnten hier ein Bewegen des Senders sein, oder eine Störquelle die sich durch die drahtlose Verbindung bewegt. Da diese Zwischenstufen durch die obigen beiden Fälle weitgehend mitabgedeckt werden wird im Folgenden auf sie nicht mehr gesondert eingegangen werden.

## 2.3 Existierende FEC Codes

label

$E_b/N_o$ [dB]	SNR als Faktor	Fehlerrate
13 dB	20	5%
10 dB	10	9%
7 dB	5	17%
6 dB	4	20%
4.77 dB	3	25%
3 dB	2	33%
2 dB	1.59	39%
1 db	1.26	44%
0 dB	1	50%

Tabelle 2.2: Umrechnungstabelle  $E_b/N_o$  in Fehlerrate

Im folgenden werden existierende FEC-Codes vorgestellt. Bei Messungen wird die Fehlerhäufigkeit in der Übertragung meist durch das normalisierte Verhältnis von empfangenen korrekten Daten zu empfangenen fehlerhaften Daten angegeben ( $E_b/N_o$ ). Da die Angabe in Dezibel

## *Kapitel 2 Grundlagen*

gewöhnungsbedürftig ist, ist in 2.2 eine Umrechnungstabelle in den normalen Faktor und in die Fehlerrate angegeben.

### 2.3.1 Viterbi Code

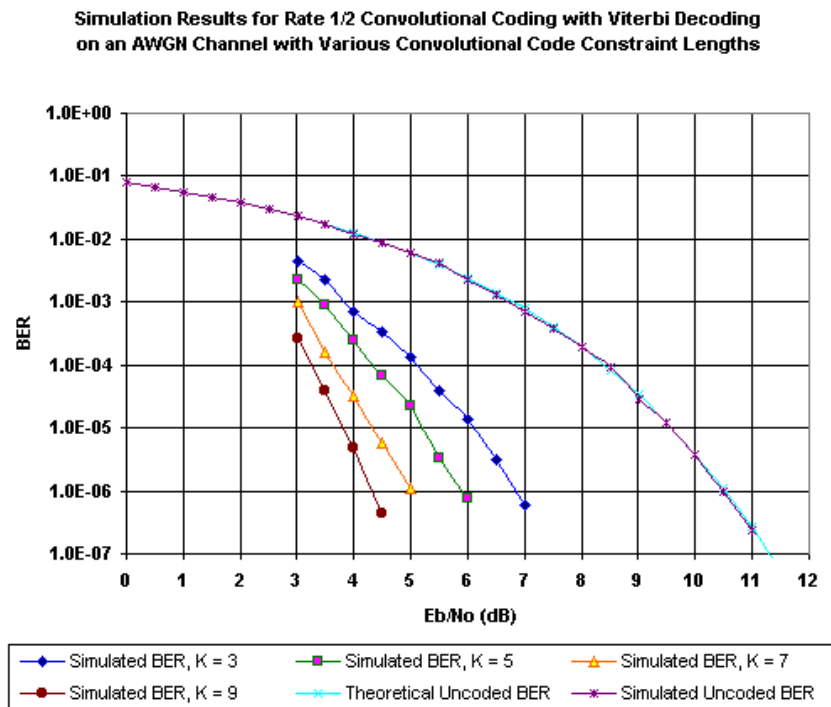


Abbildung 2.2: Messung der Korrektur-Rate eines Viterbi-Codes mit Datenrate 1/2 (aus [3])

Der nach seinem Entdecker ernannte Viterbi-Code wurde 1967 entwickelt [18]. Seitdem hat er bei Raummissionen, bei Handys oder im 802.11 WLAN-Standard Verwendung gefunden.

Der Viterbi-Code ist ein Faltungscode. Zum Generieren eines FEC-Zeichens werden dabei  $k$  Daten-Zeichen verrechnet;  $k$  kann hierbei als Parameter für die Komplexität des Verfahrens betrachtet werden. Das Encodieren kann dabei als Zustandsmaschine aufgefaßt werden, wobei die jeweils generierten Codewörter die Zustände darstellen: Abhängig vom aktuellen Codewort/Zustand, und der gerade anliegenden Datenfolge, wird zum nächsten Codewort/Zustand gewechselt. Auf der Empfängerseite wiederum wird mittels eines Hidden-Markov-Modells versucht, die eingehenden Codewörtern/Zuständen in die zugehörigen Datenfolgen zu übersetzen. Sollte ein empfangene Zustandsfolge nicht korrekt sein, so wird stattdessen die nächstwahrscheinliche Zustandsfolge angenommen. Dieser Vorgang ist anhand eines Fall-Beispiels in 2.3 dargestellt. Dabei ist die Komplexität des Decodierens linear abhängig von der Zahl  $k$  der beim Encodieren in ein FEC-Zeichen einfließenden Daten-Zeichen.

Da beim Dekodieren immer eine Folge von Zuständen/Codewörtern betrachtet wird ist dieser Algorithmus anfällig für Burst-Fehler. Auch bleibt die Korrekturrate hinter neueren Codes wie Turbo-Code oder LDPC zurück, so dass er auch bei Noise nur bedingt empfehlenswert ist.

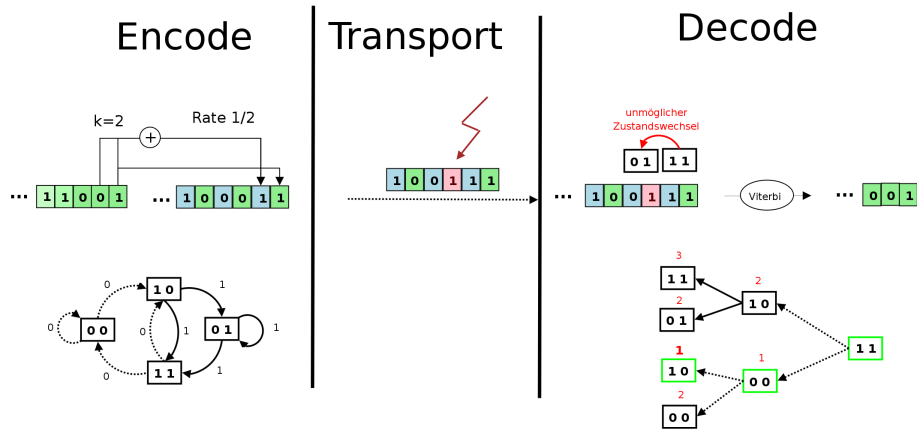


Abbildung 2.3: Schematische Darstellung des Viterbi-Algorithmus

### 2.3.2 Reed-Solomon Code

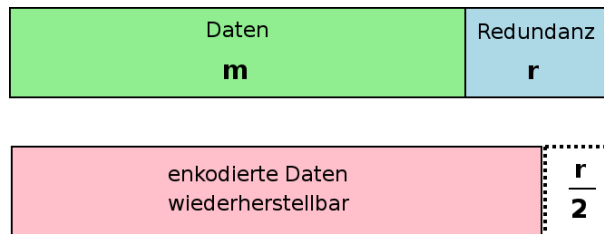


Abbildung 2.4: Der Reed-Solomon-Algorithmus' kann FEC-Zeichen/2 Fehler sicher ausgleichen

Beim *Reed-Solomon Code* handelt es sich um einen Block-Code, der Daten-Block als Polynom interpretiert, und mittels Lagrange-Interpolation weitere Stützstellen errechnet (die redundanten Daten). Die garantierte Korrekturrate des Reed-Solomon Codes entspricht der halben Länge der redundanten Daten.

Der Reed-Solomon Code wurde 1960 entwickelt [11], fand aber erst 1980 in einem serienproduzierten Produkt Verwendung (der Compact Disc). Er ist einer der derzeit verbreitetsten Codes, und findet etwa bei der Fehlerkorrektur bei Speichermedien (CDs, DVDs, BlueRay) oder beim Digital Video Broadcast (DVB) und beim Digital Audio Broadcast (DVA) Verwendung.

Ein Nachteil des Reed-Solomon ist, dass die Korrekturleistung allein von der Zahl der betrachteten Daten-Zeichen, und der Zahl der generierten FEC-Zeichen abhängt. Es gibt keinen Parameter um darüber hinaus die Komplexität des Algorithmus' zu regeln.

Die garantierte Korrekturrate des Reed-Solomon-Codes entspricht dabei der halben Redundanz. Eine gängige Datenrate ist dabei 223/255; in diesem Block könnten entsprechend 16 Zeichen korrigiert werden. Dies macht den Reed-Solomon-Code zum Beheben von Burst-Fehlern interessant. Allerdings bleibt seine Korrekturrate hinter neueren Algorithmen wie dem Turbo

Code oder dem Low-Density-Parity-Check-Code zurück; gerade bei starkem Noise wird das zum Problem.

### 2.3.3 Turbo Code

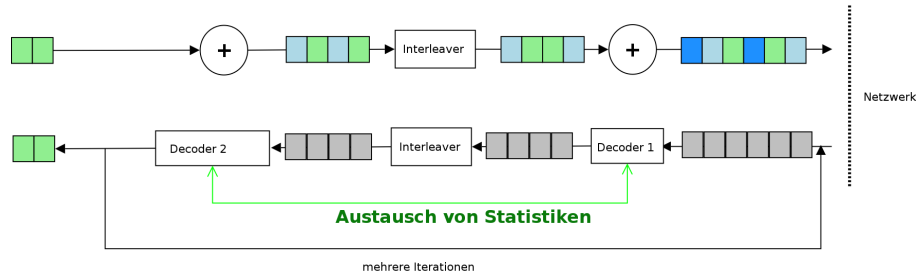


Abbildung 2.5: Schematische Darstellung des Turbo Code-Prinzips

Bei Turbo-Codes werden mehrere einfache FEC-Algorithmen (etwa Viterbi oder Reed-Solomon) hintereinandergeschaltet, wobei ein Interleaver die Zwischen-Buffer nach einem festen Schema neu ordnet. Wesentlicher Bestandteil ist, dass sich beim Dekodieren die verschiedenen Dekoder untereinander austauschen, und so arbiträre Stellen in den zu rekonstruierenden Daten klären können. Je öfter dieser Schritt wiederholt wird (die Zahl der *Iterationen*), desto besser ist die erzielte Korrekturrate (siehe 2.6). Hierbei ist es sogar möglich eine Korrekturrate nah am Shannon-Limit zu erreichen [1]. Durch die Interleaver ist der Codec auch gut gegen Burst-Fehler abgesichert.

Turbo Codes wurden 1993 von Berrou, Glavieux und Thitimajshima entwickelt [1], und finden seitdem etwa im 3G Handy-Standard, dem IEEE 802.16 WLAN-Standard, oder bei Weltraummissionen (wie dem Mars Reconnaissance Orbiter) Verwendung.

### 2.3.4 Low-Density-Parity-Check Code

Der *Low-Density-Parity-Check (LDPC) Code* besteht aus mehreren *Paritätsgleichungen*, die in einer Paritätsmatrix zusammengefaßt werden. Dabei ist kennzeichnend, dass die Matrix nur dünn besetzt ist, sprich nur wenige Zeichen eines Wortes in einer Paritätsgleichung miteinander verglichen werden. Hierbei läßt sich die Komplexität über die Dichte der Matrix steuern; ein gängiges Schema ist dabei, dass eine Paritätsgleichung fünf Daten-Zeichen umfaßt, und jedes Daten-Zeichen in drei Paritätsgleichungen vorkommt.

LDPC wurde 1963 von Galager entwickelt ([4]), ließ sich mit den damaligen Systemen nur schwer umsetzen, und wurden aber erst 1996 “wiederentdeckt” ([6]). Auch er erlaubt eine Korrekturrate nah am Shannon-Limit (siehe 2.7). Da alle Zeichen eines Blocks zueinander in Beziehung gesetzt werden ist auch gut geeignet um Burst-Fehler auszugleichen.

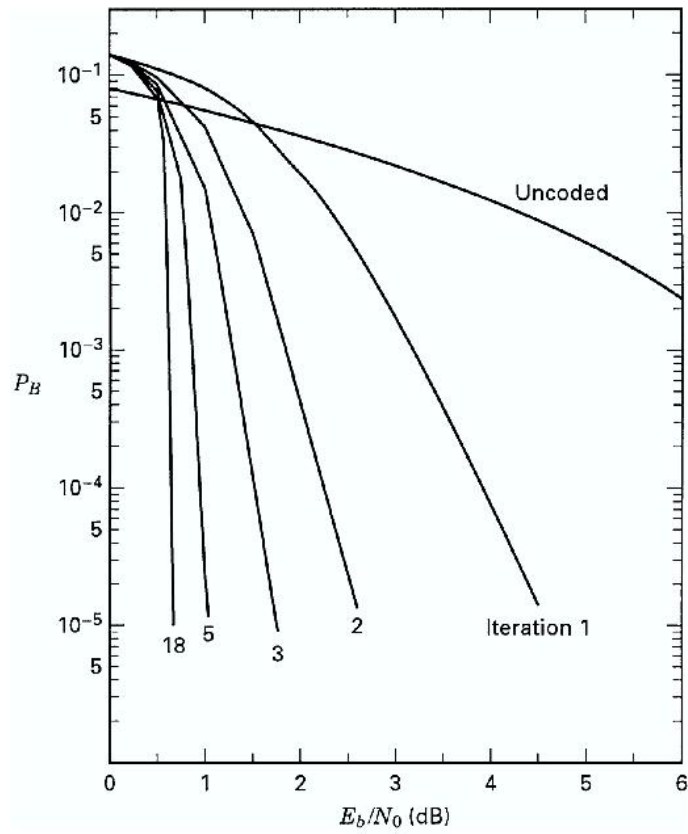


Abbildung 2.6: Korrektur-Rate eines Turbo-Codes (aus [13])

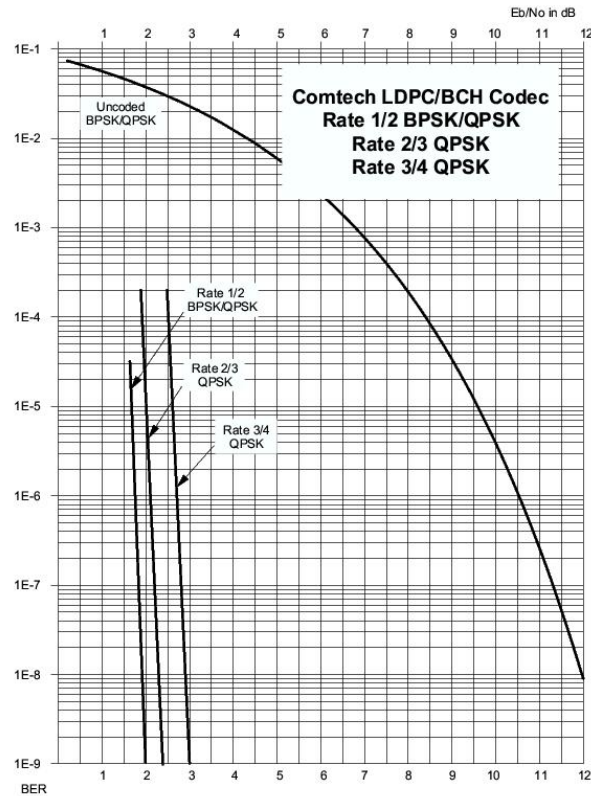


Abbildung 2.7: Korrektur-Rate eines kommerziellen LDPC-Codes in Satelliten-Modems (aus [7])

LDPC findet beispielsweise beim Digital Video Broadcast - Satellite - Generation 2 (DVB-S2) Anwendung, und konnte sich bei der Ausschreibung gegen diverse Turbo-Codes durchsetzen. Auch beim G.hn-Standard, einer Heimnetz-Technologie, die beispielsweise Datenaustausch über Strom- und Telefonkabel ermöglicht, wurde LDPC verschiedenen Turbo-Codes vorgezogen.

## 2.4 Codec-Auswahl

### 2.4.1 Entscheidung für LDPC

Beim Vergleich der Messdaten läßt sich erkennen, dass Turbo-Code und LDPC eine deutlich bessere Korrekturrate erlauben als Viterbi-Code und Reed-Solomon-Code (siehe hierzu auch Grafik 2.8. Turbo-Code und LDPC sind beide in der Lage Burst-Fehler auszugleichen.

Beide erlauben es die Komplexität über Parameter zu Steuern: die Matrixdichte bei LDPC, die Zahl der Iterationen beim Turbo-Code.

Beide finden im Bereich Multimedia in Industrie-Standards Verwendung, wobei sich hier der LDPC derzeit durchzusetzen scheint.

Den Ausschlag gab letztlich, dass für LDPC eine frei verfügbare, leistungsfähige C++-Library zur Verfügung steht, während sich für Turbo-Code nur proprietäre Angebote finden ließen.

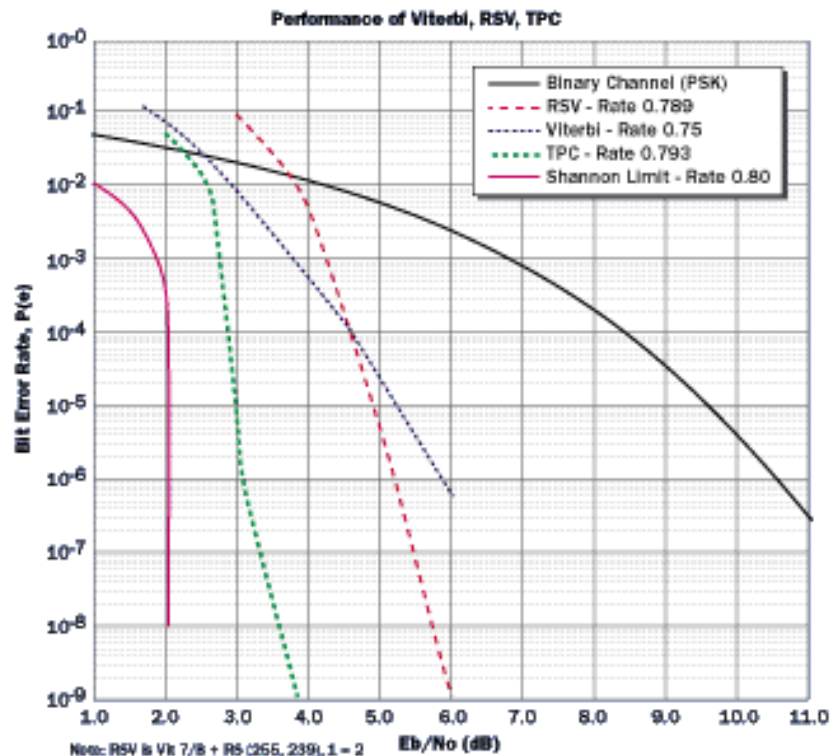


Abbildung 2.8: Korrektur-Rate von Viterbi, Viterbi-ReedSolomon und Turbo-Code, aus [19]

## 2.4.2 LDPC FEC Codec von Planète-BCAST

Für die prototypische Implementierung eines FEC Codecs wurde sich für den Low-Density-Parity-Check-Code entschieden, welcher von der Planète-BCAST-Arbeitsgruppe am Institut National de Recherche en Informatique et en Automatique (INRIA) entwickelt wurde [9]. Verwendet wird die Version 2.1 vom September 2006.

### Vorteile des Codecs

Ein wesentlicher Vorteil dieses Codecs ist seine Ausrichtung auf die Anwendungsebene. Die zu encodierenden Datenblöcke können mehrere Megabyte groß sein, was den Codec zum Absichern von Multimedia-Daten sehr geeignet macht. Bei großen Datenmengen reduziert dies nicht nur

den Overhead, es bietet vor allem auch einen besseren Schutz vor Burst-Fehlern. Darüber hinaus ist der Codec auf das Korrigieren von Paketverlusten ausgerichtet: Die Daten und der Error-Code werden in Zeichen unterteilt. Beim Dekodieren werden aus den Paketen iterativ die Ursprungsdaten rekonstruiert, wobei der Algorithmus erkennt wenn die Ursprungsdaten wiederhergestellt sind, und keine weitere Auswertung mehr nötig ist. Die Reihenfolge der Pakete spielt dabei keine Rolle.

Laut den Angaben der Entwickler erreicht die Version 1.8 des Codecs auf einem Pentium IV/3.06GHz unter Linux folgende Geschwindigkeiten. Dabei wurden 20.000 kByte Daten um 10.000 kByte Error-Code erweitert, was einer Datenrate von 2/3 entspricht; die Paketgröße lag bei 1 kByte. Bei LDPC Staircase wurden der Error-Code mit einer Geschwindigkeit von 734,0 Mbit/s erzeugt, was einer Bandbreite für das gesamte Signal von 2.205,0 Mbit/s entspricht. Für das Decodieren wird eine Geschwindigkeit von 816,5 Mbit/s angegeben. Bei LDPC Triangle wurden der Error-Code mit einer Geschwindigkeit von 443,8 Mbit/s erzeugt, was einer Bandbreite für das gesamte Signal von 1,331.4 Mbit/s entspricht. Für das Decodieren wird eine Geschwindigkeit von 434,9 Mbit/s angegeben.

Weitere Vorteile dieses Codecs sind: Dieser Codec ist unter der GNU/LGPL-Lizenz frei verfügbar und nicht durch Patente geschützt. Er wird als objektorientierte C++-Bibliothek zur Verfügung gestellt und beinhaltet zwei verschiedene Implementierungen des LDPC-Algorithmus: LDPC Staircase und LDPC Triangle. Die Bibliothek ist sowohl für Linux als auch Windows geeignet.

### Abschätzen der garantierten Korrekturrate

Mit dem Codec werden die Quellen für die Demo-Programme *simple\_coder* und *simple\_decoder* ausgeliefert. Mit diesen lässt sich eine einfache Client-Server-Verbindung aufbauen, mit *simple\_decoder* als Server. Der *simple\_coder* erzeugt daraufhin Testdaten, encodiert diese und schickt die Pakete in zufälliger Reihenfolge an den *simple\_decoder*, welcher daraus die Testdaten rekonstruiert.

Beim Verwenden von 50 kBytes Date, einer Datenrate von 2/3 und einer Zeichengröße von 1 kByte wurden zwischen 52 und 59 der 75 Pakete benötigt um die Testdaten wiederherzustellen. Dies entspricht einer garantierten Korrekturrate von 21.33%. Dies entspricht in etwa einem stark beeinträchtigen WLAN-Netz.

# Kapitel 3

## Netzwerk-Integrierte Multimedia-Middleware

Die Netzwerk-Integrierte Multimedia-Middleware (NMM) ist am Lehrstuhl für Computergraphik der Universität des Saarlandes entwickelt worden, unter der Leitung von Prof. Dr. Slusalek (REF). Als Middleware ist sie zwischen Anwendungs-Ebene und Betriebssystem-Ebene angesiedelt, und bietet die Möglichkeit auf die über das Netzwerk verteilten System-Ressourcen zuzugreifen. Dabei bieten die Schnittstellen zu den anderen Ebenen ein notwendiges Maß an Abstraktion. So kann die Middleware auf mehreren Linux-, Windows- und Mac OS X-Systemen laufen, und der Anwendungsebene die auf diesen Rechnern vorhandenen Multimedia-Ressourcen zur Verfügung stellen, ohne dass dazu auf Anwendungsebene Detailwissen um die verwendeten Betriebssysteme, oder die verwendeten Multimedia-Geräte nötig wäre. Anders als bei traditionellen Server-Client-Lösungen wird auch die Netzwerk-Ebene abstrahiert, so dass Anwendungen auf über das Netz verteilte Ressourcen zugreifen und diese kontrollieren können, als wären sie lokal.

Die NMM gibt es sowohl in einer kostenlosen Open Source Version, als auch in einer von der Motamas GmbH vertriebenen Fassung. Im folgenden werden die für diese Arbeit bedeutenden Bestandteile von NMM vorgestellt, was vor allem Netzwerk-Aspekte umfaßt. Eine gute Einführung in NMM bietet [14], eine umfassende Beschreibung von NMM findet sich in [5].

### 3.1 Flussgraph

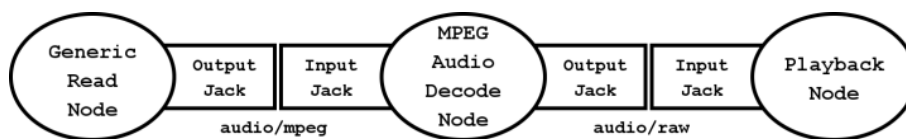


Abbildung 3.1: Darstellung eines Flussgraphen zum Abspielen von mp3s

Die NMM-Architektur ist modular: sie besteht aus verknüpfbaren Komponenten (*Knoten*), welche die Multimedia-Funktionalität kapseln. Jeweils zwei Knoten werden über eine (*Kante*)

verbunden, so dass ein *Flussgraph* entsteht. In den Kanten befinden sich *Kommunikationskanäle*, welche die Verbindungen darstellen.

### 3.1.1 Knoten

Bei den Knoten kann es sich sowohl um Hardware-Ressourcen (wie beispielsweise eine Kamera) als auch Software-Ressourcen kapseln (wie etwa das Auslesen einer Datei). Dabei werden anhand ihrer Rolle im Flußgraphen folgende Knotenarten unterschieden:

- *Quellen* senden Daten und verfügen daher über einen Ausgang
- *Senken* empfangen Daten und verfügen daher über einen Eingang
- *Filter* verarbeiten Daten, wobei sich das Format nicht ändert (Ein- und Ausgang)
- *Konverter* verarbeiten Daten, wobei sich das Format ändert (Ein- und Ausgang)
- *Demuxer* teilt Daten in mehrere Ausgänge auf (ein Eingang, mehrere Ausgänge)
- *Muxer* fügen Daten aus mehreren Eingängen zusammen (mehrere Eingänge, ein Ausgang)
- *MuxDemuxer* verfügen über mehr als einen Ein- und Ausgang

### 3.1.2 Jacks

Verbunden werden die Knoten über ihre Ein- und Ausgänge (*Jacks*). Dabei lassen sich den Jacks Formatvorgaben zuweisen, so dass sichergestellt werden kann, dass der Datenstrom auch den Anforderungen des Knotens genügen. Beispiele für Formatvorgaben wären der Typ des Datenstroms (etwa audio/mpeg für mpeg-codierte Audio-Daten, oder audio/raw für uncodierte Audio-Daten), aber auch spezifische Eigenschaften wie die Sampling-Rate oder die Zahl der Audio-Kanäle.

## 3.2 Kommunikation

In einem aktiven Flussgraph ist immer mindestens eine Quelle und eine Senke enthalten, so dass es ein *Multimedia-Datenstrom* mit eindeutiger Flussrichtung entsteht.

### 3.2.1 Nachrichten

Damit NMM-Knoten miteinander kommunizieren können müssen Nachrichten (*Messages*) versandt werden. Hier unterscheidet NMM zwei Arten von Nachrichten:

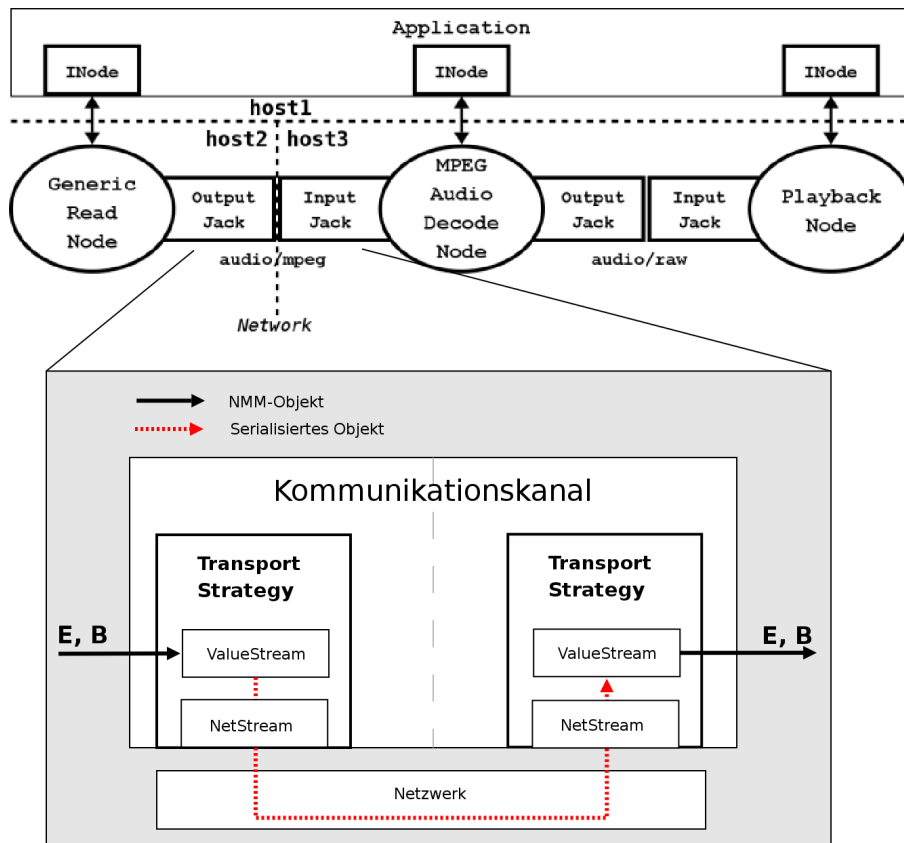


Abbildung 3.2: Darstellung der Verbindung zweier Jacks

## Buffer

*Buffer* werden verwendet, um besagte Multimedia-Inhalte zwischen den Knoten zu übertragen. Sie enthalten sowohl die Multimedia-Inhalte an sich, als auch eventuell benötigte Zusatzinformationen (wie etwa Zeitstempel/*Timestamps* zur synchronen Wiedergabe).

## CompositeEvents

Zusätzlich zu den Buffern gibt es noch *Events*, welche zum Steuern der Knoten verwendet werden. Diese werden gebündelt in Form von *Composite-Events* (*CEvents*), die aus einem oder mehreren Events bestehen, übertragen. Zum einen gibt es hier in-stream Events, welche stromabwärts (*downstream*) oder stromaufwärts (*upstream*) weitergereicht werden. Ein Beispiel könnte hier sein, downstream die nachfolgenden Knoten über eine Formatänderung zu informieren, oder upstream eine anderes Format zu fordern. Zusätzlich gibt es auch noch *out-of-band* Events, mit der die Anwendung direkt auf die Knoten zugreifen kann. Diese sind aber für das Thema dieser Arbeit von nachrangiger Bedeutung.

### 3.3 Transportstrategien

Um die Kommunikation zwischen Knoten so allgemein und modular wie möglich zu halten, werden je nach Bedarf *Transportstrategien* zusammengestellt. Dabei wird ein Paar aus Sender-Transportstrategie und Empfänger-Transportstrategie als *Binding* bezeichnet. Eine Transportstrategie ist dabei aus verschiedenen *Mechanismen* zusammengesetzt, zu denen *Serialisierungs-Mechanismen* und *Transportmechanismen* gehören können. Die Datenweitergabe innerhalb der Transportstrategien ist als eine als eine Folge von hintereinander geschalteten *Stream*-Klassen umgesetzt.

#### 3.3.1 Serialisierungs-Streams

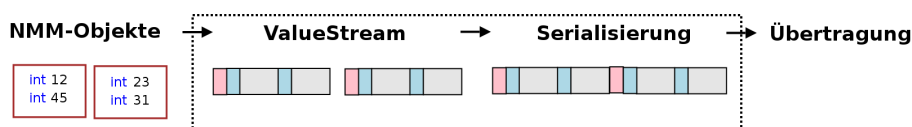


Abbildung 3.3: Serialisierung: Ein NMM-Objekt wird in mit Indizes versehene Binärdaten umgewandelt, welche dann ans Netzwerk weitergereicht werden

Bevor NMM-Nachrichten über das Netzwerk übertragen werden können, müssen sie erst in eine fortlaufende Folge von Binärdaten umgewandelt werden (*Serialisierung*), bevor diese dann von anderen Streams weiterverarbeitet und übertragen werden können. Auf diese Weise können Messages von einem Knoten an den anderen übertragen werden, welcher sie dort wieder aus den Binärdaten rekonstruiert (*Deserialisierung*). Analog zu C++-Streams wird hier zwischen ausgehenden Streams (OStreams) und eingehenden Streams (IStreams) unterschieden.

Das Serialisieren erfolgt durch *ValueStreams*, welche ein abstraktes Objekt in eine mit Typinformationen versehene Datenfolge umwandeln.

Diese Binärdaten lassen sich dann von folgenden Streams weiterverarbeiten. Dies wäre der Punkt, an dem etwa der Einbau von FEC-Funktionalität ansetzen würde.

#### 3.3.2 Netzwerk-Streams

Anschließend können diese Binärdaten von den *NetStreams* als Datenpakete durch entsprechende Transportprotokolle übertragen werden.

# Kapitel 4

## Integration von FEC in NMM

### 4.1 Anwendungsszenario

Die Integration der FEC-Funktionalität als Stream-Klassen innerhalb einer Transportstrategie bietet sich aufgrund mehrerer Faktoren an:

- Die Modularität und Austauschbarkeit der Stream-Klassen erlaubt es vielfältige Anwendungsszenarien abzudecken, etwa durch das Hintereinanderschalten von FEC-Algorithmen.
- Zusätzliche benötigte Funktionalität kann leicht in zusätzlichen Stream-Klassen in den Datenübertragungs-Prozess eingebunden werden.

Beim Übertragen von Datenpaketen über eine unsichere Verbindung können dabei zwei Fehlerarten auftreten:

- *Bitfehler in den Datenpaketen*: Der Inhalt eines Pakets wird durch Störungen verändert. Die gängigen Transportprotokolle (UDP, TCP, RTP) verwenden Checksummen in den Paketen um solche Veränderungen zu entdecken. Bei UDP wird in einem solchen Fall das Paket nicht weiter ausgewertet (*gedropped*) - die enthaltenen Daten kommen auf der Empfängerseite nicht. (Wie in [16] ausgeführt bieten diese Checksummen keinen vollständigen Schutz, allerdings ist der Anteil an Pakete mit unentdeckten Bitfehlern (1 in 16 Mio. bis 1 in 10 Mrd.) sehr gering.)
- *Verlust von Datenpaketen*: Durch solche erkannten Bitfehler, aber auch durch länger anhaltende Unterbrechungen der Verbindung, kann es zum Verlust ganzer Pakete kommen. Gerade bei drahtlosen Datenübertragung ist es nicht selten, dass gleich mehrere Pakete in Folge verloren gehen (*Burst-Fehler*).

Deshalb liegt der Schwerpunkt dieser Arbeit auf dem Wiederherstellung verlorenen Datenpakete. Um das zu erreichen ist es grundlegend, dass der enkodierte Datenblock mehrere Pakete umfaßt, so dass durch Paket-Verluste verlorenen Daten durch die redundanten Daten in anderen Paketen rekonstruiert werden können.

Im Rahmen dieser Arbeit ist eine Rekonfiguration der FECs zur Laufzeit (etwa das Erhöhen der Komplexität oder der Datenrate) nicht vorgesehen. Prinzipiell bietet NMM jedoch Anwendungen die Möglichkeit, die Transportstrategie anzupassen.

## 4.2 Aufbau einer FECTransportStrategy

Unter diesen Gesichtspunkten setzt sich die neugeschaffene *FECTransportStrategy* wie folgt zusammen:

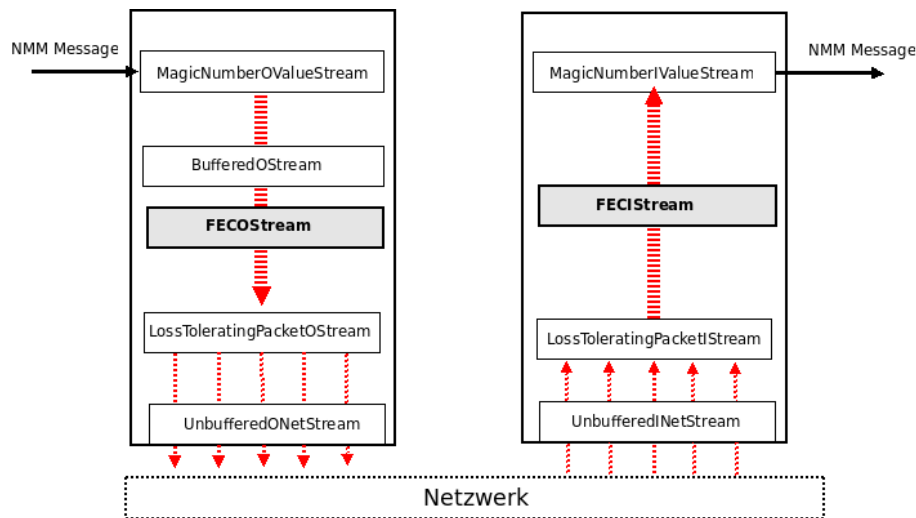


Abbildung 4.1: Einbau von FEC in die Transportstrategie: Das Encodieren findet zwischen dem Buffern der serialisierten Daten und dem Senden ans Netzwerk statt

Der *MagicNumberOValueStream* serialisiert die eingehenden NMM-Objekte. Aus diesen Daten erzeugt der *BufferedOStream* Datenblöcke fester Größe. Diese sind auf den nachfolgenden *FECStream* abgestimmt. Der *FECStream* liefert einen Buffer, der sowohl Ursprungsdaten als auch redundante Daten enthält. Dieser Signalbuffer wird dann vom *LossToleratingPacketOStream* in nummerierte Pakete aufgespalten, welche dann einzeln vom *UnbufferedONetStream* über das Netzwerk an den *UnbufferedINetStream* auf Empfängerseite übertragen werden. Der *LossToleratingPacketIStream* rekonstruiert daraus den Signalbuffer, wobei die geordnete Datenübertragung durch die Nummerierung der Pakete sichergestellt ist. Sollten Pakete verloren gehen, so werden die entsprechenden Stellen im Buffer durch Leerdaten ersetzt. Aus diesem Signalbuffer versucht der *FECStream* die Ursprungsdaten zu rekonstruieren, anhand derer der *MagicNumberIValueStream* dann die entsprechenden NMM-Objekte erzeugt.

## 4.3 Test-Streams

Um die Fehlerkorrektur auch ohne eine verlustbehaftete Netzwerkverbindung testen zu können werden zusätzlich zu den oben genannten Streams noch drei *ErrorStreams* bereitgestellt:

- Den *BitErrorStream*, welcher mit einem bestimmten Prozentsatz an Bytes verfälscht. Welche Bytes im Buffer betroffen sind wird dabei zufällig bestimmt.

- Den *BlockErrorStream*, welcher den Buffer in Blöcke einer wählbaren Byte-Länge unterteilt, und dann für jeden Block anhand einer frei wählbaren Wahrscheinlichkeit bestimmt, ob dieser auf NULL gesetzt wird. Mit diesem Stream können auf einfache Weise Blockfehler bzw. Paketverluste approximiert werden.
- Den *PacketLossErrorStream*, welcher mit einer frei wählbaren Wahrscheinlichkeit den ihm übergebenen Buffer “verliert”, indem er ihn nicht an seinen Partnerstream weiterreicht. Dieser Stream ist vor allem zur Verwendung nach dem LossToleratingPacketStream gedacht: Der PacketStream wandelt eingehende Buffer in mehrere Pakete um, von denen der PacketLossErrorStream einen bestimmten Prozentsatz zufällig verliert.

# Kapitel 5

## Implementierung

Im folgenden Kapitel wird auf die Implementierung der Streams und des Testprogramms eingegangen. Die Stream-Klassen unterscheiden sich hierbei in ausgehende Streams (5.1) und eingehende Streams (5.2). Alle Streams verfügen über Methoden zur Datenweitergabe (sogenannte *Serialisierungs-Operatoren*), wobei sie sich stark an der Syntax von C++-Streams orientieren. Von besonderem Interesse für die Datenweitergabe zwischen Streams sind hierbei die Methoden *writeBuffer(Buffer\*, Size)* für OStreams, und *readBuffer(Buffer\*, Size)* für IStreams.

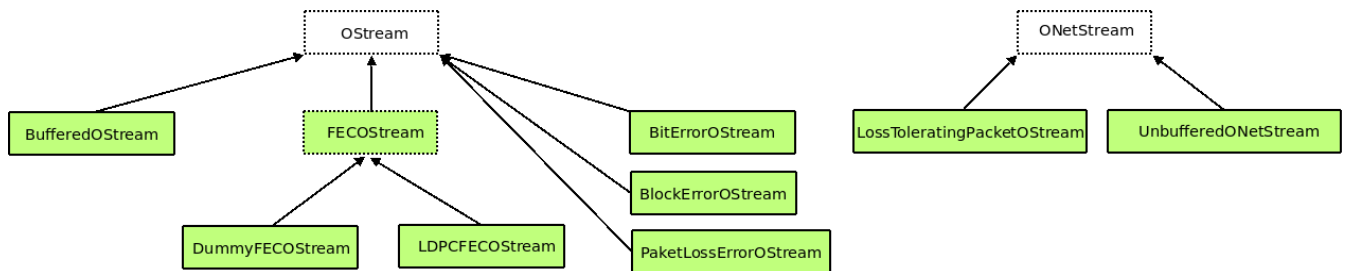


Abbildung 5.1: Ableitungs-Hierarchie der ausgehenden Streams

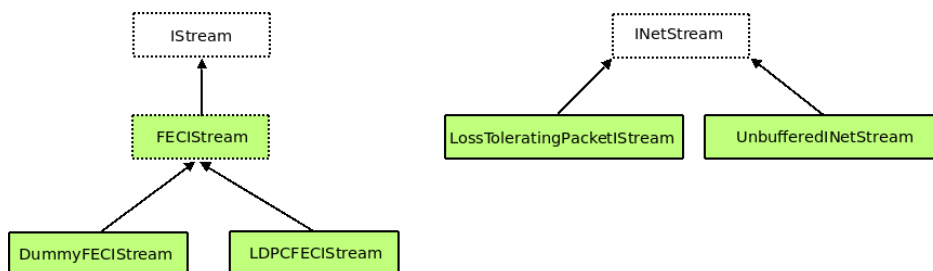


Abbildung 5.2: Ableitungs-Hierarchie der eingehenden Streams

## 5.1 FECStream

*FECStream* ist die Klasse, von der alle Streams, die FEC-Funktionalität implementieren, ableiten. *FECStream* selbst leitet von *Stream* ab. Entsprechend implementiert *FECStream* die Methoden zur Datenweitergabe Serialisierungs-Operatoren. Die Grundidee ist, dass die Klasse über die Serialisierungs-Operatoren Daten erhält, diese enkodiert bzw. dekodiert, und dann das Ergebnis weiterreicht. Dabei arbeitet der Encoder bzw. Decoder mit festen Buffergrößen, welche nicht mit den durch die Serialisierungs-Operatoren erhaltenen Buffern übereinstimmen müssen.

Der Ablauf bei einem *FECStream* sieht dabei folgendermaßen aus:

- Von außen wird ein Serialisierungs-Operator (etwa *writeBuffer*) aufgerufen, der einen *DataBuffer\** und dessen Größe übergibt.
- Der Serialisierungs-Operator wiederum ruft *internalWriteBuffer(DataBuffer\*, Size)* auf, welches den Buffer auf die für den Encoder benötigte Größe anpaßt: Ist der eingehende Buffer größer, so wird er über mehrere *InternalDataBuffer* aufgeteilt. Ist er zu klein, so wird er in einen neuen *InternalDataBuffer* kopiert und mit '-' aufgefüllt.
- Diese *InternalDataBuffer* werden dann an *FECEncode(InternalDataBuffer\*, SignalBuffer\*)* gegeben. Die Implementierung von *FECDecode* erfolgt in abgeleiteten Klassen, welche dann einen *SignalBuffer* erzeugen.
- Im Anschluss wird der *SignalBuffer* mittels *writeBuffer* an den *PartnerStream* übergeben.

*FECStream* folgt einem ähnlichen Schema:

- Es wird von außen Serialisierungs-Operator (beispielsweise *readBuffer*) aufgerufen, welche einen zu füllenden *DataBuffer* samt Größe übergeben.
- Der Serialisierungs-Operator ruft die Methode *internalReadBuffer(SignalBuffer\*, Size)* auf. Diese fordert dann wiederum selbst mittels *readBuffer* einen *InternalSignalBuffer* vom *PartnerStream* an.
- Dieser *InternalSignalBuffer* wird dann zum Dekodieren an *FECDecode(InternalDataBuffer\*, InternalSignalBuffer\*)*: Anhand der im Buffer enthaltenen FEC-Daten werden die beschädigte Daten wiederhergestellt. Die konkrete Implementierung liegt auch hier wieder bei den abgeleiteten Streams.
- Im Anschluss wird der *InternalDataBuffer* dann in den zu füllenden *DataBuffer* kopiert. Auch hier wird entsprechend gesplittet wenn der *InternalDataBuffer* zu groß ist. Ist der *InternalDataBuffer* hingegen zu klein, so wird auf weitere Daten gewartet.

### Einschränkung

Dass *FECStream* bei zu kleinen Buffern Zusatzzeichen ('-') einfügt geschieht auch beim Senden eines leeren Buffers (als Zeichen des Verbindungsabbruchs). Wird kein *ValueStream* verwendet, so kann dies zum Anfügen von Zusatzzeichen am Ende eines Datentransfers führen. Hierzu ist noch kein Workaround implementiert.

### 5.1.1 DummyFECStream

Der *DummyFECStream* leitet vom *FECStream* ab. Er soll vor allem zum Testen des Frameworks dienen. Er implementiert keinen wirksamen FEC, stattdessen wird beim Encodieren der Signalbuffer mit Füllbuchstaben aufgefüllt; beim Decodieren werden sie wieder entfernt. Außerdem werden die Buffer auf cout ausgegeben, um so das Debuggen zu erleichtern.

### 5.1.2 LDPCFECStream

Im *LDPCFECStream* wurde der in 2.4.2 vorgestellte Low-Density-Parity-Check-Codec integriert.

#### **FECEncode(DataBuffer\*, SignalBuffer\*)**

Das Encodieren erfolgt in folgenden Schritten:

- Zuerst wird eine neue *LDPCFecSession* mittels *InitSession* initialisiert, wobei die Paketgröße, und die Zahl der Daten- und der Error-Code-Pakete übergeben wird und das Encoder-Flag gesetzt wird.
- Der *DataBuffer* wird in den *SignalBuffer* kopiert. Zu diesem gibt es ein Pointer-Array *SymbolCanvas\**, welches den *SignalBuffer* in paketgroße Stücke partitioniert.
- Im letzten Schritt wird für jedes zu generierende FEC-Paket einmal die *LDPCFecSession*-Methode *BuildParitySymbol(SymbolCanvas\*, CurrentFECPacketPointer\*)* aufgerufen.

#### **FECDecode(DataBuffer\*, SignalBuffer\*)**

Das Decodieren erfolgt ähnlich:

- Es wird wieder eine *LDPCFecSession* angelegt, diesmal mit gesetztem Decoder-Flag.
- Der *SignalBuffer* wird in einen *SignalBufferCanvas\** aufgeteilt; ein leerer *DataBuffer* analog in einen *SignalBufferCanvas\**
- Jetzt werden iterativ aus den einzelnen *SignalPaketen* die *DatenPakete* rekonstruiert: *DecodingStepWithSymbol(DataSymbolCanvas\*, CurrentSignalPacketPointer\*, DecodeStep)*

## 5.2 Support Streams

### 5.2.1 BufferedOStream

Der *BufferedOStream* ermöglicht es, Daten bis zu einer bestimmten Blockgröße zu sammeln. Auf diese Weise kann verhindert werden, dass der nachfolgende Stream auch ineffizient kleine Datenmengen sofort verarbeitet. Ein Beispiel wäre hier der *FECStream*, der alle eingehenden Datenblöcke sofort encodiert.

## 5.2.2 LossToleratingPacketStream

Der *IgnorantOPacketStream* spaltet einen Buffer in eine feste Zahl Pakete auf, welche dann einzeln an nachfolgende Buffer weitergegeben werden. Die Pakete setzen sich aus einer Paketnummer und den Nutzdaten zusammen. Dies erlaubt es, dem *LossToleratingIPacketStream* die Pakete wieder geordnet zum ursprünglichen Buffer zusammenzusetzen. Die Implementierung dieser der Mechanik wurde vom *PacketNetStream* übernommen. Anders als beim *PacketNetStream* wird ein Buffer jedoch nicht verworfen, wenn einzelne Pakete den Empfänger nicht erreichen. Stattdeseb werden die unvollständigen Bereiche mit Nullen aufgefüllt, und dieser fehlerhafte Buffer an den nachfolgenden Stream weitergegeben.

## 5.2.3 UnbufferedNetStream

Der *UnbufferedNetStream* basiert auf dem *BufferedNetStream*. Wie dieser dient er dazu, Daten mittels des UDP-Transportprotokolls zu übertragen. Anders als der *BufferedNetStream* werden kleine Datenblöcke jedoch nicht vor dem Senden zu größeren Buffern zusammengefaßt.

## 5.3 Error Streams for Testing

### 5.3.1 BitErrorStream

Der *BitErrorStream* leitet direkt von *Stream* ab. Er wird mit einer Fehler-Wahrscheinlichkeit (Integer von 0 bis 100) initialisiert. Bei Erhalt eines neuen Buffers wird zufällig bestimmt, ob in diesem Buffer ein Fehler auftreten soll. Ist dies der Fall, dann werden die letzten 6 Bit XOR(101010) genommen.

### 5.3.2 PacketLossErrorStream

Der *PacketLossErrorStream* verfügt wie der *BitErrorStream* über eine Fehler-Wahrscheinlichkeit. Tritt diese beim Erhalt eines Buffers ein, so wird dieser nicht weiterpropagiert.

## 5.4 fec\_test Testanwendung

Zum Testen eines einfachen FEC-Szenarios wurde die Linux-Anwendung *fec.test* implementiert. Hierbei handelt es sich um ein Client-Server-Setup, das auf zwei getrennten Rechnern gestartet wird. Beim Starten bauen Client und Server jeweils ihre Transportstrategie auf, anschließend beginnt der Client Daten aus einer Datei auszulesen, gibt diese an seine Transportstrategie, welche sie letztlich an den Server überträgt. Dort durchlaufen die Daten dessen Transportstrategie, und werden schließlich in eine Datei ausgegeben. Dies wird dazu genutzt, die Daten auf Client-Seite zu Enkodieren und daraufhin Fehler einzufügen. Auf Serverseite wird anschließend versucht die ursprünglichen Daten durch Decodieren wieder herzustellen.

```
Usage:
.fec_test -r [options]
.fec_test -w -H host [options]
Options:
-f : test file to be decoded
-o : output file to store decoded data
-t : udp, tcp (udp is default)
-r : Reads from network on the -p specified port
-w : Writes to -H host on the -p specified port
-H : Sets the host of the distributed server registry
-p : Sets the port of the distributed server registry
-T : Connection timeout on receiver side in ms (Standard: 10000ms)
-C : FEC codec to be used: LDPC, dummy
-v : increase verbosity (may be given multiple times)
-h : Show this help
```

Abbildung 5.3: Parameter von fec\_test

Zusätzlich messen sowohl Client als auch Server die Datenmenge und die Zeit, die die Daten zum Durchqueren der jeweiligen Transportstrategie benötigen, um daraus die Bandbreite der Transportstrategie zu berechnen.

# Kapitel 6

## Testen der Implementierung

### 6.1 Testumgebung

Alle Tests wurden am NMM-Labor des Lehrstuhls Computergraphik der Universität des Saarlandes durchgeführt. Im Labor stehen mehrere Rechner mit folgender Ausstattung zur Verfügung:

- AMD X2 4200+ (Dual-Core, je 2,2 GHz)
- 2GB Arbeitsspeicher
- 2 Gigabit Netzwerkkarte
- Betriebssystem: Ubuntu Linux (Kernel 2.6.28)

Die Rechner sind durch SMCCGS16-Switch der Firma SMC Networks verbunden, welcher über ausreichend Bandbreite verfügt [15].

### 6.2 Fehlersimulation mittels BlockErrorStream

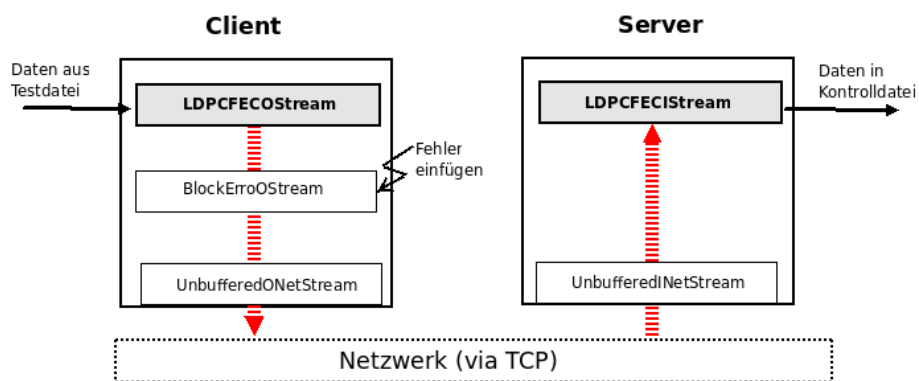


Abbildung 6.1: Transportstrategien von Client und Server

## 6.2.1 Starten des Testprogramms

Zum Testen des LDPC-Codexs wird die unter 5.4 beschriebene Beispielanwendung *fec\_test* verwendet.

Dabei soll folgender Datenfluss stattfinden (siehe 6.1):

- Die Daten werden auf Client-Site aus einer Testdatei ausgelesen und an den *LDPCFECOSTream* gegeben
- Der *LDPCFECOSTream* enkodiert die Daten mittels LDPC. (Bei einer Paketgröße von 8 Bytes Dabei beträgt die Datenrate 2048/3072).
- Der *BlockErrorOSTream* fügt Blockfehler mit einer von der Anwendung angegebenen Wahrscheinlichkeit ein. (Dabei entspricht ist Blocklänge ebenfalls 8 Bytes.)
- Der *BufferedErrorOSTream* schickt die Daten dann via TCP zum BufferedErrorIStream auf Server-Seite, welcher sie an den *LDPCFECOSTream* weiterreicht
- Der *LDPCFECOSTream* enkodiert die Daten.
- Anschließend werden die Daten in eine Kontrolldatei geschrieben.

Dabei wird *fec\_test* zuerst auf dem als Server fungierenden Rechner gestartet (der hier den Hostnamen *tone* hat), und anschließend auf dem Client (hier *patch*), der dann beginnt Verbindung aufbaut und die Daten an den Server überträgt.

Zum Starten des Servers wird *tone* folgender Befehl verwendet:

```
.fec_test -r -t tcp -C LDPC -o Flash-HDSpot_Fehlerwahrscheinlichkeit
```

Erläuterung der Parameter auf Server-Seite:

- *-r*: Wie “read” - Das Programm wird als Server gestartet
- *-t tcp*: Als Transportprotokoll wird TCP verwendet. Dies reduziert zwar die Übertragungsgeschwindigkeit, allerdings kann so der unkontrollierte Verlust von Daten (etwa durch Überschreiten der Bandbreite) ausgeschlossen werden. In diesem Fall wird der BufferedINetStream zum Empfangen der Daten vom Netzwerk verwendet.
- *-C LDPC*: Es wird der LDPCFECIStream zum Dekodieren der Daten verwendet.
- *-o FlashHDSpot\_Fehlerwahrscheinlichkeit*: Die Dekodierten Daten werden in der Datei *FlashHDSpot\_Fehlerwahrscheinlichkeit* abgelegt, wobei *Fehlerwahrscheinlichkeit* hier analog zu den Einstellungen auf Client-Seite gewählt wird.

Der Aufruf des Clients auf *patch*:

```
.fec_test -w -H tone -t tcp -C LDPC -T 300 -f Flash-HDSpot  
-E Fehlerwahrscheinlichkeit
```

Erläuterung der Client-Parameter:

- *-w*: Wie “write” - Das Programm wird als Server gestartet.
- *-H*: Der Host, mit dem die Verbindung aufgebaut werden soll. (Wird nicht mit *-p* ein Port angegeben, so wird der Port 22801 verwendet.)

- *-t tcp*: TCP als Transportprotokoll, es wird der BufferedONetStream zum Versenden der Daten benutzt.
- *-C LDPC*: Die Daten werden mittels LDPCFECStream enkodiert.
- *-T 300*: Nach 300 Sekunden wird die Verbindung zum Server beendet. Dies ist vor allem interessant wenn man die Programme per Skript startet, um so dem Server genug Zeit zu geben alle Daten auszuwerten und sich zu beenden.
- *-f Flash-HDSpot*: Es wird die Datei Flash-HDSpot übertragen. Hierbei handelt es sich um eine 9283856 Bytes (8,85 MBytes) große Videodatei (H.264/AVC mit 1280 x 720 Auflösung (HD), 30 fps; 35s Laufzeit).
- *-E Fehlerwahrscheinlichkeit*: Hier wird die Wahrscheinlichkeit eines Paketverlustes angegeben (zwischen 0 und 100 Prozent). Dieser Wert wird an den BlockErrorOStream weitergegeben. Pakete sind dabei standardmäßig 8 Bytes groß, was dem Standardwert des LDPCFECStreams entspricht.

## 6.2.2 Ergebnis

### Messergebnis

Der Server berechnet für jede empfangene Datei die Bandbreite: Die insgesamt in die Kontrolldatei geschriebene Datenmenge geteilt durch die Zeit vom Empfangen des ersten Buffers bis zum Schreiben des letzten Buffers.

Die Zahl der fehlerhaften Bytes wurde unter Verwendung des Programms *cmp* errechnet, welches zwei Dateien byteweise vergleicht. Mit der Option *-l* gestartet gibt es die Position jedes abweichenden Bytes in eine neue Zeile aus; die Zeilen können dann mit dem Programm *wc* gezählt werden:

```
cmp -l Flash-HDSpot Flash-HDSpot.Fehlerrate | wc -l
```

Das Ergebnis der Messungen findet sich in Tabelle 6.1.

**Einschränkung** Wie in 5.1 beschrieben kann es ohne Verwendung eines *ValueStreams* dazu kommen, dass Zusatzzeichen am Dateiende angefügt werden ('-', bis zu ein FEC-Buffer lang). Um diese zu entfernen wird das Programm *split* verwendet, welches eine Datei in mehrere splittet:

```
split -b 9283856 Flash-HDSpot.Fehlerrate Flash-HDSpot.Fehlerrate_
```

Dies erzeugt eine Datei *Flash-HDSpot.Fehlerrate\_aa*, welche dann mit *Flash-HDSpot* verglichen werden kann.

## Bewertung

Die Bandbreite liegt mit durchgehend ein MByte/s deutlich über den Anforderungen der HD-Testdatei. Das beschränkende Element war hier die Verbindung über TCP; mittels UDP ließen sich vermutlich ein noch höherer Durchsatz erreichen.

Bei einer Fehlerrate bis zu 6% korrigiert der LDPC-Codec alle Fehler; bis zu einer Fehlerrate von 15% sind praktisch keine Störungen sichtbar. Ab 19% nimmt die Qualität jedoch spürbar ab; ab 22% ist das Video fast durchgehend gestört.

Kontrolldatei	Bandbreite	Fehlerhafte Bytes
Flash-HDSpot_00	1.08743 MByte/s	0
Flash-HDSpot_01	1.08855 MByte/s	0
Flash-HDSpot_02	1.10893 MByte/s	0
Flash-HDSpot_03	1.2038 MByte/s	0
Flash-HDSpot_04	1.19254 MByte/s	0
Flash-HDSpot_05	1.11624 MByte/s	0
Flash-HDSpot_06	1.07757 MByte/s	0
Flash-HDSpot_07	1.13654 MByte/s	80
Flash-HDSpot_08	1.08484 MByte/s	64
Flash-HDSpot_09	1.12713 MByte/s	0
Flash-HDSpot_10	1.03872 MByte/s	111
Flash-HDSpot_11	1.08075 MByte/s	620
Flash-HDSpot_12	1.17617 MByte/s	128
Flash-HDSpot_13	1.08216 MByte/s	940
Flash-HDSpot_14	1.10078 MByte/s	1243
Flash-HDSpot_15	1.10283 MByte/s	167
Flash-HDSpot_16	1.09081 MByte/s	1541
Flash-HDSpot_17	1.04846 MByte/s	2012
Flash-HDSpot_18	1.17106 MByte/s	2627
Flash-HDSpot_19	1.13811 MByte/s	5021
Flash-HDSpot_20	1.15106 MByte/s	10349
Flash-HDSpot_21	1.05888 MByte/s	16864
Flash-HDSpot_22	1.04953 MByte/s	35145
Flash-HDSpot_23	1.09602 MByte/s	82093
Flash-HDSpot_24	1.05578 MByte/s	177378
Flash-HDSpot_25	1.03166 MByte/s	365123
Flash-HDSpot_26	1.07978 MByte/s	647304
Flash-HDSpot_27	1.12334 MByte/s	993645
Flash-HDSpot_28	1.0795 MByte/s	1393489
Flash-HDSpot_29	1.2313 MByte/s	1806667
Flash-HDSpot_30	1.21385 MByte/s	2233477

Tabelle 6.1: Messergebnis für LDPC, Fehlersimulation mittels BlockErrorOStream

# Kapitel 7

## Zusammenfassung und Ausblick

### 7.0.3 Erreichte Ziele

Das Ziel dieser Arbeit bestand darin, Vorwärtsfehlerkorrektur in NMM zu integrieren. Hierzu wurden existierende FEC-Codecs analysiert. NMM wurde um *FECStreams* erweitert, welche in Kombination mit anderen Streams (teils vorhanden, teils implementiert) ein modulares und erweiterbares Framework zum Nutzen von Vorwärtsfehlerkorrektur bereitstellen. Es wurde prototypisch eine ausgewählte FEC-Library eingebunden (*LDPCFECStream*). Zur Simulation von Fehlern wurden mehrere Streams bereitgestellt, und es wurde ein Testprogramm geschrieben um die Tauglichkeit des FEC-Einbaus zu testen.

Hierbei hat sich gezeigt, dass Vorwärtsfehlerkorrektur zu einem deutlichen Qualitätsgewinn führt, und das zu einer Bandbreite, die zum Übertragen von Multimedia-Daten tauglich ist.

### 7.0.4 Erweiterungsmöglichkeiten

#### Einbau eines weiteren Codecs für hohe Fehlerraten

Der verwendete LDPC-Codec liefert bereits gute Ergebnisse, verliert jedoch ab einer Fehlerrate von 20% deutlich an Wirkung und bleibt in diesem Bereich hinter den Erwartungen zurück. Hier würde es sich anbieten, einen FECStream mit einer anderen Bibliothek (TurboCode oder wieder LDPC) in einen FECStream zu implementieren.

#### Integration in bestehende NMM-Anwendung

Die Integration in bestehende NMM-Anwendungen wie *clit* oder die *MMBox* steht noch aus. Hierzu müßten unter anderem entsprechende Transportstrategien zur Verwendung in Graph-Descriptions implementiert werden.

#### Anpassung der FECs zur Laufzeit

NMM bietet die Möglichkeit, Transportstrategien zur Laufzeit zu ändern, was bisher für FECs nicht genutzt.

### **Automatisches Anpassen der FECs an Störungsmuster**

Das Automatische Anpassen der FECs an Kriterien wie die zu erwartende Fehlerrate würde es ermöglichen Ressourcen wie Bandbreite besser zu nutzen. Hier gibt es bereits diverse Methoden um das Channel-Verhalten vorherzusagen, beispielsweise die Prognose anhand von Markov-Ketten [2], den *USF*-Algorithmus [8] oder den *Optimal Progressive Error Recovery Algorithm* [10].

# Literaturverzeichnis

- [1] BERROU, GLAVIEUX, T. Near shannon limit errorcorrecting coding and decoding Turbo-codes. *IEEE International Communications Conference, Proceedings* (1993).
- [2] CHEN, CHUA, T. U. N. Adaptive error coding using channel prediction. *Wireless Networks* 5 (1999).
- [3] FLEMING, C. A tutorial on convolutional coding with viterbi decoding. <http://home.netcom.com/%7Echip.f/viterbi/simrslts.html> (2006).
- [4] GALLAGER, R. *Low Density Parity Check Codes*. M.I.T. Press Monograph, 1963.
- [5] LOHSE, M. *Network-Integrated Multimedia Middleware, Services, and Applications*. PhD thesis, Saarbrücken, 2005.
- [6] MACKAY, D. J., AND NEAL, R. M. *Near Shannon Limit Performance of Low Density Parity Check Codes*. Electronics Letters, 1996.
- [7] MILLER, R. New forward error and modulation technologies. <http://tinyurl.com/comtech-ldpc> (2005).
- [8] PADHYE, CHRISTENSEN, M. U. C. A new adaptive fec loss control algorithm for voice over ip applications. *Proceedings of IEEE International Performance, Computing and Communication Conference* (2000).
- [9] PLANETE RESEARCH TEAM, I. Ldpc fec. <http://planete-bcast.inrialpes.fr> (2006).
- [10] QAISAR, R. Optimal progressive error recovery for wireless sensor networks using irregular ldpc codes. *CISS Paper 80* (2007).
- [11] REED, S. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* (1960).
- [12] SHANNON, C. E. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.
- [13] SKLAR, B. Fundamentals of turbo codes. [http://ptgmedia.pearsoncmg.com/images/art\\_sklar3\\_turbocodes/elementLinks/art\\_sklar3\\_turbocodes.pdf](http://ptgmedia.pearsoncmg.com/images/art_sklar3_turbocodes/elementLinks/art_sklar3_turbocodes.pdf).
- [14] SLUSALLEK, LOHSE, R. U. W. Networkintegrated multimedia middleware (nmm). *Proceedings of ACM Multimedia 2008 ACMMM2008* (2008).
- [15] SMC. Smcgs16 datasheet. [http://www.smc.com/files/AL/ds\\_GS16\\_GS24.pdf](http://www.smc.com/files/AL/ds_GS16_GS24.pdf).
- [16] STONE, P. When the crc and tcp checksum disagree. *ACM SIGCOMM* (2000).
- [17] THIELEN, M. *Design and Development of a Quality of Service Framework for the Network-Integrated Multimedia Middleware (NMM)*. PhD thesis, Saarbrücken, 2007.
- [18] VITERBI, A. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*. 13, Nr. 2 (1967).

## *Literaturverzeichnis*

- [19] WILLIAMS, D. Turbo-charging next-gen wireless, optical systems.  
*<http://www.commsdesign.com/story/OEG20010119S0060>* (2001).

# Anhang A

## Quellcode

Im Rahmen dieser Arbeit wurden folgende Dateien implementiert:

BufferedStream:

```
/nmm/comm/serialize/BufferedOStream.cpp  
/nmm/comm/serialize/BufferedOStream.hpp
```

LossToleratingPacketStream:

```
/nmm/comm/netstrategy/fecsupport/LossToleratingPacketOStream.cpp  
/nmm/comm/netstrategy/fecsupport/LossToleratingPacketOStream.hpp  
/nmm/comm/netstrategy/fecsupport/LossToleratingPacketIStream.cpp  
/nmm/comm/netstrategy/fecsupport/LossToleratingPacketIStream.hpp
```

UnbufferedNetStream:

```
/nmm/comm/netstrategy/fecsupport/UnbufferedONetStream.cpp  
/nmm/comm/netstrategy/fecsupport/UnbufferedONetStream.hpp  
/nmm/comm/netstrategy/fecsupport/UnbufferedINetStream.cpp  
/nmm/comm/netstrategy/fecsupport/UnbufferedINetStream.hpp  
/nmm/comm/netstrategy/fecsupport/Makefile.am
```

FEC Streams:

```
/nmm/comm/serialize/fec/FECStream.cpp  
/nmm/comm/serialize/fec/FECStream.hpp  
/nmm/comm/serialize/fec/FECIStream.cpp  
/nmm/comm/serialize/fec/FECIStream.hpp
```

```
/nmm/comm/serialize/fec/DummyFECStream.cpp  
/nmm/comm/serialize/fec/DummyFECStream.hpp  
/nmm/comm/serialize/fec/DummyFECIStream.cpp
```

## *Anhang A Quellcode*

/nmm/comm/serialize/fec/DummyFECIStream.hpp

/nmm/comm/serialize/fec/LDPCFECOSTream.cpp

/nmm/comm/serialize/fec/LDPCFECOSTream.hpp

/nmm/comm/serialize/fec/LDPCFECIStream.cpp

/nmm/comm/serialize/fec/LDPCFECIStream.hpp

/nmm/comm/serialize/fec/Makefile.am

/nmm/comm/serialize/fec/LDPC/Makefile.am

/nmm/comm/serialize/fec/LDPC/src/Makefile.am

### **Error Streams:**

/nmm/comm/serialize/fecsupport/BitErrorOStream.cpp

/nmm/comm/serialize/fecsupport/BitErrorOStream.hpp

/nmm/comm/serialize/fecsupport/PacketLossErrorOStream.cpp

/nmm/comm/serialize/fecsupport/PacketLossErrorOStream.hpp

/nmm/comm/serialize/fecsupport/BlockErrorOStream.cpp

/nmm/comm/serialize/fecsupport/BlockErrorOStream.hpp

/nmm/comm/serialize/fecsupport/Makefile.am