Hochschule für
Technik und Wirtschaft
Dresden
University of Applied Sciences

Masterarbeit

in der Studienrichtung Intelligente Informations- und
Kommunikationstechnologien
im Studiengang Angewandte Informationstechnologien
der Fakultät Informatik/Mathematik

Betrachtung von ratenlosen Codes und prototypische Implementierung von Online Codes

Eingereicht von: Martin Dönicke
Eingereicht am: 9. Dezember 2013
Betreuender Hochschullehrer: Prof. Dr.-Ing. Jörg Vogt

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

3GPP 3rd Generation Partnership Project
AL-FEC Application Layer-Forward Error Correction
AB Auxiliary Block – Zusatzblock
ARQ Automatic Repeat reQuest
BEC Binary Erasure Channel
CB Check Block – kodierter Block
CM Composite Message – zusammengesetzte Nachricht aus SBs und ABs
FEC Forward Error Correction
IP Internet Protocol
IPTPS International Workshop on Peer-To-Peer Systems
LDPC Low-Density Parity-Check
LT Luby Transform
LTE Long Term Evolution
PRNG Pseudo Random Number Generator
Raptor Rapid Tornado
RS Reed-Solomon
SB Source Block – Quellblock
TCP Transmission Control Protocol
UDP User Datagram Protocol

1 Einleitung

1.1 Motivation

In die heutigen Praxis haben bereits viele Forward Error Correction (FEC)-Verfahren Einzug gehalten, jedoch beruhen die meisten eingesetzten Techniken sehr selten auf den neuesten entwickelten Verfahren. Das erste Dokumente zur Theorie von ratenlosen Codes erschien bereits im Jahr 2002 (siehe [LMS+97]), jedoch ist der praktische Einsatz besagter Codes aktuell nur selten vorzufinden.

Durch dauerhaft ansteigende Datenmengen, welche täglich im Internet transportiert werden, steigt der Bedarf an Datendurchsatz sowohl von Datenanbietern als auch von Endanwendern. Oft ist es der Fall, dass eine Vielzahl von Empfängern die selben Daten verlangen. Ein Vorgehen wie bei klassischen Broad-/Multicast-Transfers ist hier jedoch nicht möglich, da die Menge der Empfänger die Daten nicht zur selben Zeit anfordern. Ein Beispiel hierfür sind Softwareupdates für Smartphones, Computer oder andere Entertainment-Geräte. Die Spieleplattform Steam bietet es beispielsweise an, aufwändige Computerspiele vollständig über das Internet zu beziehen, wodurch täglich enorme Datenmengen entstehen. Der verursachte Traffic durch diesen Vertriebsweg bewegt sich in Deutschland innerhalb einer Woche bereits im einstelligen Petabyte-Bereich (siehe [Cor13]). Auch bei der Aktualisierung von Kartendaten für Navigationsgeräte trifft das eben beschriebene Szenario zu, wenn auch mit einer geringeren Datenmenge. Hinzu kommt jedoch, dass die Datenverbindung möglicherweise sehr instabil ist und somit effiziente Mechanismen zur Fehlerkorrektur verwendet werden müssen.

In dieser Arbeit soll gezeigt werden, ob sich ratenlose Codes gewinnbringend in solchen praktischen Szenarien einsetzen lassen. Hierfür wird nach einer Betrachtung von Luby Transform (LT) Codes, Rapid Tornado (Raptor) Codes und Online Codes ein direkter Vergleich dieser vorgenommen. Im praktischen Teil der Arbeit wird die entstandene prototypische Implementierung von Online Codes aufgearbeitet und ausgewertet. Anhand dieser werden Aussagen über die Tauglichkeit des Verfahrens für verschiedene Anwendungsfälle getroffen.

1.2 Aufbau

Der erste Teil der Arbeit befasst sich mit der Betrachtung von ratenlosen Codes.

Zu Beginn werden kurz einige Grundlagen zusammengefasst, welche für das Verständnis der nachfolgenden Ausarbeitung benötigt werden.

Im Anschluss daran wird ein kurzer Einblick zum direkten Vorgänger der ratenlosen Codes gegeben.

Nachfolgend werden die verschiedenen Arten von ratenlosen Codes erläutert, hierbei wird insbesondere auf Online Codes eingegangen, da diese elementarer Bestandteil des zweiten Teils der Arbeit sind.

Ein direkter Vergleich der Verfahren schließt den ersten Teil der Arbeit ab.

Der zweite Teil der Arbeit befasst sich mit der während des Bearbeitungszeitraums entstandenen prototypischen Implementierung von Online Codes.

Zunächst werden Anforderungen, Entscheidungen und daraus resultierende Lösungen in der Entwurfsphase betrachtet.

Danach wird auf ausgewählte Aspekte der Implementierung eingegangen.

Den Teil abschließend wird eine Leistungsbewertung von Online Codes und der prototypischen Implementierung vorgenommen.

Zum Abschluss der Arbeit werden die Ergebnisse zusammengefasst und ein Ausblick gegeben.

> Anmerkung: Die Abbildungen A.1 bis A.15 befinden sich im Anhang der Arbeit.

2 Betrachtung von ratenlosen Codes

2.1 Grundlagen

Mithilfe von FEC ist es möglich Informationen über einen fehleranfälligen Kanal fehlerfrei zu übertragen. Fehler können durch das Hinzufügen von Redundanz zu den Informationen erkannt oder sogar korrigiert werden. Hieraus ergibt sich der Begriff der **Coderate -** R,

$$R = \frac{k}{n}$$

k ... Anzahl übertragener Informationsbits
n ... Anzahl übertragener Datenbits

welche das Verhältnis von Informationsbits zu Datenbits beschreibt. Je geringer die Coderate, desto geringer der Informationsanteil in einem übertragenen Datum.

Daraus lässt sich die **Anzahl der Paritätsbits -** p

$$p = n - k$$

ableiten. Mithilfe dieses frei wählbaren Parameters ist es möglich eine Codierung an einen Kanal anzupassen. Mit einer steigenden Fehleranfälligkeit muss p entsprechend erhöht werden, woraus eine Verringerung der Coderate R folgt.

Bei der heutigen Kommunikation über das Internet via Internet Protocol (IP) kann man vereinfacht von einer Übertragung über einen Binary Erasure Channel (BEC) ausgehen (nach [RU08]). Hierbei werden Nachrichten entweder fehlerfrei übertragen oder der Empfänger erkennt, dass ein Fehler aufgetreten ist, womit er die Nachricht verwerfen kann. Letzterer Fall ist einem Nachrichtenverlust gleichzustellen, worauf bereits der Name BEC(zu deutsch „binärer Auslöschungskanal“) hinweist. Das Übermitteln einer fehlerhaften Nachricht ist in diesem Modell ausgeschlossen.

Die maximale Menge an Informationen, welche über den Kanal fehlerfrei übertragen werden kann, wird als **Kanalkapazität -** C

$$C = 1 - P_e$$

P_e ... Fehlerwahrscheinlichkeit

bezeichnet. Im Jahr 1948 veröffentlichte Shannon die Arbeit „A mathematical theory of communication“ (siehe [Sha01]). Aus ihr geht hervor, dass es nicht nötig ist, eine Coderate weit unter der Kanalkapazität zu verwenden, um eine zuverlässige Kommunikation zu ermöglichen. In Theorem 9 ist die heutzutage sogenannte (siehe [Sno01]) **Shannon-Grenze**

$$\frac{C}{R} - \epsilon$$

definiert. Weiterhin wird gesagt, dass es ausreicht, Informationen mit einer Coderate knapp unter dieser Grenze zu kodieren, um sie ausreichend zuverlässig zu übertragen. Voraussetzung hierfür ist jedoch eine ausreichend große Länge der zu übertragenden Daten.

Das Coupon Collector's-Problem (zu deutsch „Sammelbilderproblem“) beschreibt einen Sachverhalt der Wahrscheinlichkeitstheorie. Es folgt ein vereinfachtes Beispiel, der Bezug zur Arbeit wird später hergestellt.
Angenommen ein Sammelbilderalbum besteht aus 100 Sammelbildern, von denen alle gleich verteilt sind. Die Bilder sind einzeln verpackt und vor dem Kauf weiß man nicht, welches Bild man kauft. Wie viele Bilder muss man im Schnitt kaufen um von jedem Sammelbild mindestens eins zu besitzen?

$$n \cdot \sum_{k=1}^{n} \frac{1}{k}$$

Daraus folgt, dass man bei einer Menge von 100 Sammelbildern im Durchschnitt $100 \cdot \sum_{k=1}^{100} \frac{1}{k} \approx 518.7$, also 519 Bilder kaufen muss um von jedem Bild mindestens ein Exemplar zu besitzen.

2.2 Vorgänger ratenloser Codes – Tornado Codes

Tornado Codes besitzen eine feste Coderate und gelten als direkter Vorgänger der ratenlosen Codes. Die Firma Digital Fountain hält Patente innerhalb der USA für dieses Verfahren, sodass ein freier Einsatz nicht möglich ist. Die theoretische Grundlage wurde 1997 von Michael G. Luby et al. im Dokument „Practical loss-resilient codes“ (siehe [LMS+97]) veröffentlicht, wobei der Begriff „Tornado Code“ im Dokument noch nicht auftaucht. Im Abstract wird sowohl die geringe Komplexität der Codes, als auch die Schwierigkeit der mathematischen Hintergründe in Design und Analyse betont. Auf die Verfahrensweise wird nicht detailliert eingegangen, da es sich bei Tornado Codes nicht um ratenlose Codes handelt.

2.2.1 Entstehung

Tornado Codes setzen Low-Density Parity-Check (LDPC) Codes (mitunter auch als Gallager-Codes bezeichnet) ein, welche jedoch durch zeitgleiches Auftreten von Reed-Solomon (RS) Codes in den Hintergrund geraten sind. Bereits im Jahr 1963 wurde die Theorie zu LDPC Codes in [Gal63] veröffentlicht. Erst im Jahr 1997 wurde, auf diesen Codes aufbauend, die Theorie der Tornado Codes in [LMS+97] veröffentlicht.

2.2.2 Verfahrensweise

Tornado Codes basieren auf irregulären, bipartiten Graphen, welche sorgfältig gewählt werden müssen. Sie eignen sich für große Blocklängen und besitzen eine En- und Dekodierkomplexität von $O(nln(1/\epsilon))$. Dieser lineare Aufwand ist ein großer Fortschritt bezogen auf die Kodierung großer Datenmengen. Im Gegensatz zu reinen LDPC Codes wird bei Tornado Codes ein mehrschichtiger Ansatz verfolgt, wodurch der Aufwand verringert wird.

Die Kodierung erfolgt in n Schichten, wobei in den oberen $n-1$ Schichten mittels LDPC kodiert wird, in der n-ten und somit untersten Schicht jedoch mit einem RS-Code gearbeitet wird. Die Anzahl der Datenblöcke verringert sich somit je Kodierungsschicht, womit der Berechnungsaufwand für jede tiefer liegende Schicht kleiner wird. Der Vorteil hierbei ist, dass der RS-Code mit einer sehr kleinen Datenmenge im Vergleich zu den Ursprungsdaten arbeiten muss und somit trotz seiner relativ hohen Komplexität schnell berechnet werden kann. Eine schematische Darstellung der Verfahrensweise zeigt Abbildung 2.1.

Um einen Tornado Code eindeutig zu spezifizieren, müssen folgende Parameter

festgelegt werden, welche sich je nach getroffener Wahl zu einer großen Menge an Daten anhäufen können:

- Anzahl der Stufen
- Anzahl der Recovery-Blöcke pro Stufe
- Wahrscheinlichkeitsverteilung zur Erstellung der Blöcke in den $n-1$ oberen Schichten

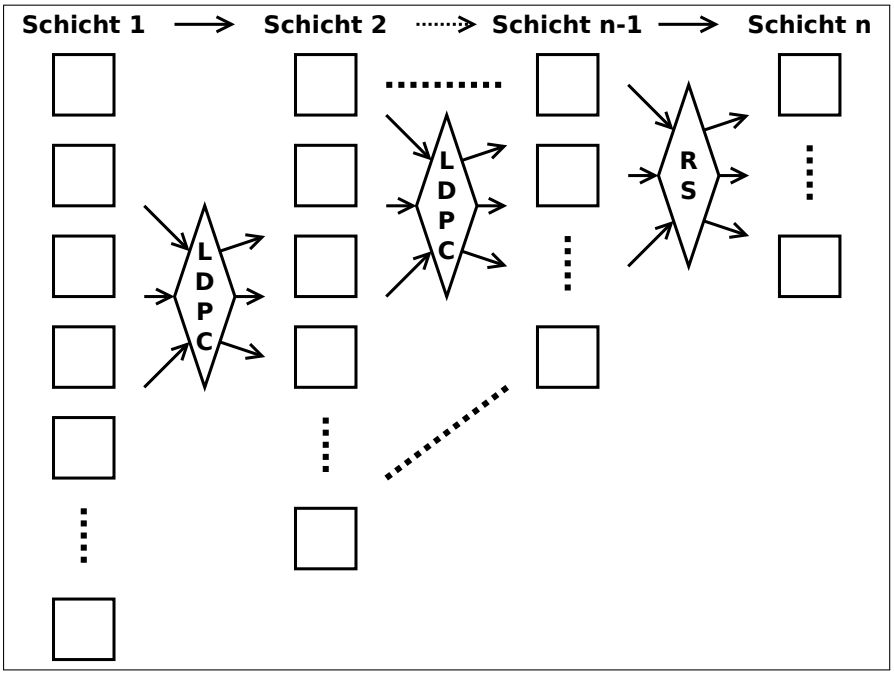

Abbildung 2.1: Verfahrensweise Tornado Codes

2.3 Stand der Technik

Mit der Veröffentlichung vom Dokument „LT codes“ (siehe [Lub02]) im Jahr 2002 entstand die Gruppe der ratenlosen Codes. Im Gegensatz zu anderen Kodierungstechniken ist es hiermit möglich, nahezu unendlich viele kodierte Daten zu generieren, unabhängig von der zugrunde liegenden Größe der zu kodierenden Daten. Der Begriff Coderate (siehe Abschnitt 2.1) der „klassischen“ Kodierungstheorie lässt sich hierbei nur bedingt verwenden, da diese variabel ist und erst nach einem erfolgreichen Dekodiervorgang ermittelt werden kann.

Bei den bisher bekannten Verfahren muss die Coderate vor dem Kodierungsprozess festgelegt werden. Mit ratenlosen Codes ist es somit möglich, solange neue Daten zu generieren und zu versenden, bis der Empfänger ausreichend Daten empfangen hat um den Dekodiervorgang erfolgreich durchzuführen. Entscheidend hierbei ist nicht die Tatsache, welche Daten empfangen wurden, sondern lediglich die Menge der empfangenen Daten. Ein demonstratives Beispiel hierfür wäre das Füllen eines Gefäßes mit Wasser unter einem Springbrunnen. Der Springbrunnen gibt kontinuierlich „Daten“ von sich, ein „Empfänger“ muss lediglich eine bestimme Menge „Daten empfangen“ um sein „Gefäß zu füllen“, wann er sein Gefäß befüllt und ob er dazwischen den Vorgang unterbricht spielt keine Rolle. Abgeleitet von diesem Beispiel wird der Begriff Fountain Code (zu deutsch „Fontänencode“) häufig als Synonym für ratenlose Codes verwendet.

Anders als bei „klassischen“ Kodierungsverfahren wird hier im Regelfall mehr als 100% der ursprünglichen Datenmenge benötigt um die Daten erfolgreich dekodieren zu können. Die benötigte Datenmenge zum erfolgreichen Dekodieren wird oft als

$$(1 + \epsilon) \cdot k$$

ϵ ... Reception Overhead
k ... Länge der Ursprungsdaten

angegeben. Der Parameter ϵ lässt sich in gewissen Grenzen anpassen, wobei folgende Faustregel gilt: je weniger Overhead, desto mehr Rechenaufwand.

LT Codes sind die ersten Codes dieser Klasse, darauf aufbauend entstanden später Raptor und Online Codes. Im Folgenden wird auf die Verfahrensweise, sowie Vor- und Nachteile der einzelnen Codes eingegangen. Online Codes werden hierbei ausführlicher beschrieben, da diese elementarer Gegenstand der prototypischen Implementierung sind, welche im zweiten Teil dieser Arbeit behandelt wird. Im

Anschluss an die Betrachtung der einzelnen Verfahren folgt eine kurze Zusammenfassung, sowie ein direkter Vergleich derer.

2.3.1 Luby Transform Codes

Bei LT Codes handelt es sich um die ersten ratenlosen Codes. Sie stellen die Grundlage für die Nachfolger Raptor und Online Codes dar. Die alleinige Anwendung des Verfahrens ist jedoch meistens nicht praktikabel, da sie einen logarithmischem Aufwand bezüglich Kodierung und Dekodierung mit sich bringen.

2.3.1.1 Verfahrensweise

Dem Verfahren liegen zwei Wahrscheinlichkeitsverteilungen für die Wahl der Blockgrade zugrunde. Kurz gefasst ist dies zum einen die eher theoretisch-optimale und nicht sehr praxistaugliche **Ideal Soliton Distribution**

$$\begin{aligned} p(1) &= \frac{1}{k} \\ p(i) &= \frac{1}{i \cdot (i-1)} \qquad \text{für } i = 2, \ldots, k \end{aligned}$$

k ... höchstmöglicher Grad, hier: Anzahl Source Blocks (SBs)

sowie die suboptimale, aber praxistaugliche **Robust Soliton Distribution**.

$$R = c \cdot ln(\frac{k}{\delta}) \cdot \sqrt{k} \quad \text{mit } c > 0$$

$$p(i) = \begin{cases} \frac{R}{i \cdot k} & \text{für } i = 1, \ldots, \frac{k}{R} - 1 \\ \frac{R \cdot ln(\frac{R}{\delta})}{k} & \text{für } i = \frac{k}{R} \\ 0 & \text{für } i = \frac{k}{R} + 1, \ldots, k \end{cases}$$

δ ... Fehlerwahrscheinlichkeit des Dekodierers
c ... frei wählbare Konstante

Nach Wahl einer Verteilung für die Kodierung kann der Datenaustausch zwischen Sender und Empfänger stattfinden. Kodiert und dekodiert wird nach den folgenden Vorschriften:

Kodiervorgang (nach [Lub02])

1. Wähle einen Grad d zufällig mittels einer zuvor festgelegten Wahrscheinlichkeitsverteilung.
2. Wähle genau d SBs zufällig aus, hierbei wird eine Gleichverteilung verwendet. Es darf jeder SB maximal einmal vorkommen.
3. Verknüpfe alle d SBs via **XOR**($\oplus$) um den Wert des Check Blocks (CBs) zu erhalten.

Dekodiervorgang (nach [Lub02])

1. Bilde alle CBs mit dem Grad 1 auf die entsprechenden SBs ab.
2. Verringere den Grad jedes CBs um die Anzahl der jeweils beinhalteten und durch Schritt 1 aufgelösten SBs.
3. Wenn alle SBs bekannt sind, ist die Dekodierung erfolgreich abgeschlossen, ansonsten gehe zum nächsten Schritt.
4. Wenn jetzt wieder CBs mit Grad 1 vorhanden sind, gehe zu Schritt 1, ansonsten wird der Dekodiervorgang abgebrochen. Wenn es möglich ist neue CBs zu beziehen, so kann der Dekodiervorgang danach erneut gestartet werden, ansonsten schlägt das Dekodieren fehl.

Abb. 2.2 zeigt ein vereinfachtes Beispiel der Zuordnung von SBs zu CBs, welche Ergebnis eines Kodiervorgangs ist. CB `a` besitzt im Beispiel Grad 3 und beinhaltet die SBs `1`, `3` und `4`. Durch Lösung des linearen Gleichungssystems lassen sich die einzelnen SBs aus den CBs rekonstruieren. Ein möglicher Dekodiervorgang könnte wie folgt aussehen:

1. Dekodiere `b` $\rightarrow$ rekonstruierte SBs: `4`
2. Dekodiere `c` $\rightarrow$ rekonstruierte SBs: `4, 1`
3. Dekodiere `d` $\rightarrow$ rekonstruierte SBs: `4, 1, 2`
4. Dekodiere `a` $\rightarrow$ rekonstruierte SBs: `4, 1, 2, 3`

Inwiefern die Metadaten (Grad und beinhaltete SBs) der CBs, zusätzlich zu den Rohdaten, dem Empfänger übermittelt werden, wird dem Anwender überlassen. In [Lub02] werden lediglich Vorschläge hierfür unterbreitet, wie z.B.

- simple Übermittlung der Werte

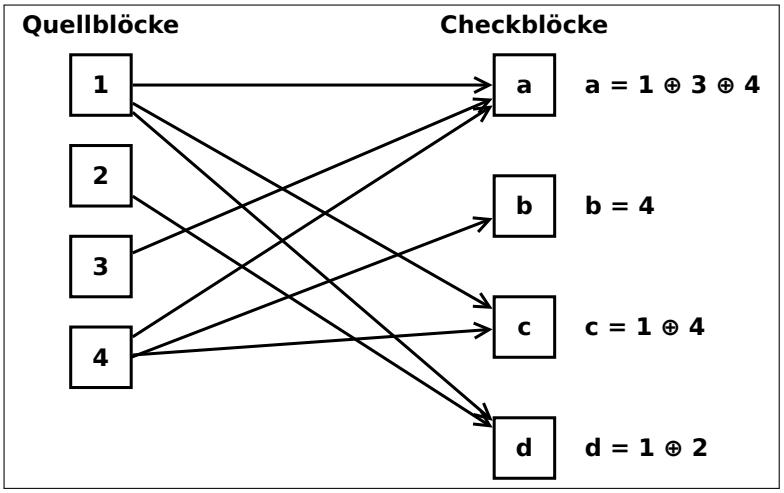

Abbildung 2.2: Beispiel: Kodiervorgang eines LT Codes

- Berechnung über Empfangszeit des CBs
- Berechnung über relative Position des CBs
- Berechnung über Empfangszeit des CBs
- Berechnung über Schlüsselwort des CBs, welches vom Kodierer zugewiesen wird

Es wird die Möglichkeit genannt, dass das Schlüsselwort zur Initialisierung eines Pseudo Random Number Generator (PRNG) genutzt werden kann, jedoch wird darauf hingewiesen, dass mögliche Methoden nicht im Fokus des Dokuments liegen.

2.3.1.2 Fazit

LT Codes sind für den praktischen Einsatz mit größeren Datenmengen nicht sehr gut geeignet, da sie eine leicht überlineare Komplexität aufweisen. Weiterhin werden LT Codes durch das Patent „US 6307487“ (siehe [Lub01]) geschützt. Inwiefern sich das Problem der leicht überlinearen Komplexität bewältigen lässt, zeigen die nächsten behandelten Verfahren, Raptor Codes und Online Codes.

2.3.2 Rapid Tornado Codes

Der wesentliche Vorteil von Raptor Codes gegenüber LT Codes ist die lediglich lineare Komplexität für Kodierung und Dekodierung. Im Jahr 2004 wurden sie mit dem Dokument „Raptor codes“ (siehe [Sho06]) publiziert. Heute gibt es mehrere standardisierte Varianten des Verfahrens, beispielsweise R10 und den Nachfolger

RaptorQ (entspricht Version 12, Namensgebung durch die Firma Qualcomm). Da Raptor Codes mit Patenten versehen sind, ist es meist nicht praktikabel diese in freien Softwareprojekten zu verwenden. Der Einsatz ist somit eher in größeren, finanziell gestützten Projekten wiederzufinden, praktische Einsatzbeispiele folgen im nächsten Abschnitt.

2.3.2.1 Verfahrensweise

Raptor Codes arbeiten zweischichtig, in der äußeren Schicht findet das Outer Encoding statt, dementsprechend in der inneren Schicht das Inner Encoding. Hierbei kommt beim Inner Encoding ein LT Code zum Einsatz, beim Outer Encoding können jedoch verschiedene Verfahren angewandt werden. Der Erfinder von Raptor Codes empfiehlt in [Sho06] eine Verkettung von Hamming-Codes gefolgt von LDPC Codes, diese Codekette ist jedoch für den Einsatz auf einem BEC optimiert. Ein allgemeiner Ansatz wäre die Verwendung eines einzelnen Codes mit einer hohen Rate, wie beispielsweise ein reiner LDPC Code (siehe [Ven12]).

Weiterhin muss bei Raptor Codes sowohl zwischen systematischen, als auch zwischen unsystematischen Varianten unterschieden werden.

> Anmerkung: Beim Einsatz von Raptor Codes können Quelldaten sowohl in Blöcke, als auch in Symbole unterteilt werden (Symbol $\subseteq$ Block $\subseteq$ Datei). Der Einfachheit halber und um Konsistenz zu den anderen Verfahren zu wahren wird im folgenden Abschnitt nur von dem Begriff Block Gebrauch gemacht. Es wird davon ausgegangen, dass die Datei genau einem Raptor-Block entspricht und dieser in mehrere Raptor-Symbole aufgeteilt wird, welche hier als Blöcke bezeichnet werden.

Bei systematischen Codes sind die `k` ersten kodierten Blöcke identisch mit den `k` Quellblöcken. Somit ist es möglich bei Erhalt der ersten `k` kodierten Blöcke die Datei ohne Overhead zu dekodieren. Anders ist das bei unsystematischen Codes, hier müssen die ersten `k` kodierten Blöcke nicht mit den `k` Quellblöcken identisch sein. Im Regelfall ist hier also ein gewisser Overhead ϵ bei der Übertragung nötig um eine Datei erfolgreich zu dekodieren, da die ersten `k` kodierten Blöcke nicht zwingend alle Quelldaten enthalten.

Im folgenden wird auf einige Besonderheiten der beiden in der Praxis am meisten eingesetzten standardisierten Verfahren eingegangen. Die Verfahrensweisen bis in das letzte Detail zu erklären steht nicht im Fokus des Dokuments, da die Verfahren sehr komplex sind (für Details siehe [Sho06], [SL11], [Ven12]). Hierbei wird jedoch noch einmal darauf hingewiesen, dass es sich bei beiden um patentierte Verfahren

handelt, was dazu beitrug Raptor Codes nicht als Grundlage für den im zweiten Teil vorgestellten Prototypen zu verwenden.

2.3.2.1.1 R10 Codes R10 Codes (abgeleitet von Raptor Version 10) liegt der RFC 5053 (siehe [LSWS07]) zu Grunde. Sie haben bereits Einzug in einige Standards gefunden.

Standardisierungen (Auszug aus [SL11])

- 3GPP Multimedia Broadcast Multicast Service (3GPP TS 26.346)
- IETF RFC 5053
- IP Datacast (DVB-IPDC) (ETSI TS 102 472 v1.2.1) for DVB-H and DVB-SH
- IPTV (DVB-IPTV) (ETSI TS 102 034 v 1.3.1) Streaming
- IPTV (DVB-IPTV) (DVB A086r7, draft ETSI TS 102 034 v 1.4.1)

Beim Mobilfunkstandard Long Term Evolution (LTE) werden sie als Application Layer-Forward Error Correction (AL-FEC) für Broadcasts eingesetzt (siehe [LPSK13]).

2.3.2.1.2 RaptorQ Codes RaptorQ Codes, entsprechend dem RFC 6330 (siehe [LSW+11]), sind die Weiterentwicklung von R10 Codes. Wesentliche Neuerungen sind:

- Blockgröße von sehr klein bis zu ca. 3,4GB wählbar
  → kleine Blockgrößen sind nützlich für Streaming, somit guter Kompromiss aus Datenschutz und Verzögerung möglich, andernfalls maximale Dekodier- und Schutzeffizienz bei großer Blockgröße (siehe [PTC13])
- Symbol-Operationen sowohl im GF(2) als auch im GF(256)
  → Möglichkeit des geringeren Reception Overheads gegenüber R10, jedoch nur geringfügig komplexer, da nur wenige Berechnungen innerhalb des GF(256)
- 256 mal mehr Encoding Symbole als bei R10 möglich, sowie ca. 7 mal mehr Symbole pro Block
  → bessere Anpassungsmöglichkeit an Streaming sowie an einfachen Dateitransfer
- Inactivation Decoding (Hybridverfahren bestehend aus Belief Propagation und Gauß-Verfahren)
  → kombiniert Optimalität des Gauß-Verfahrens mit der Effizienz des Belief Propagation (siehe [SL11])

2.3.2.2 Fazit

RaptorQ Codes sind in nahezu allen Belangen den R10 Codes überlegen, lediglich der höhere (dennoch lineare) Dekodieraufwand ist ein kleiner Nachteil der RaptorQ Codes. Jedoch kann durch diesen höheren Dekodieraufwand eine sehr niedrige Fehlerwahrscheinlichkeit des Dekodierers gewährleistet werden (siehe [PTC13]). Gründe die gegen den Einsatz von Raptor Codes sprechen sind jedoch Patente wie z.B: [SLK05] und [SL05].

2.3.3 Online Codes

Online Codes wurden von Petar Maymounkov entwickelt und deren theoretische Grundlage 2002 im Dokument „Online codes“ (siehe [May02]) veröffentlicht. Im Jahr 2003 wurden sie mit dem Paper „Rateless Codes and Big Downloads“ (siehe [MM03a]) auf dem International Workshop on Peer-To-Peer Systems (IPTPS) vorgestellt. Wobei im ersten Schriftstück sehr auf mathematische Grundlagen und Details eingegangen wird, steht im Fokus des zweiten Dokuments der praktische Einsatz von Online Codes.

Bei Online Codes handelt es sich um ratenlose Codes mit linearer Kodier- und Dekodierkomplexität, welche frei von Patenten sind und somit frei eingesetzt werden können. Im Folgenden wird detailliert auf das Verfahren eingegangen, da diese Informationen essenziell für das Verständnis des zweiten Teils der Arbeit sind.

2.3.3.1 Verfahrensweise

Es wird wie bei Raptor Codes zweischichtig mit einem Outer- und Inner Encoding gearbeitet. Das Inner Encoding ist wie bei Raptor Codes ein spezieller LT Code. Hingegen ist das Outer Encoding im direkten Vergleich relativ simpel umgesetzt. Nachfolgend wird auf die beiden Kodierschichten genauer eingegangen und im Anschluss daran noch einmal ein Überblick über den gesamten Kodier- und Dekodiervorgang gegeben. Die einzelnen Schritte sind in Abb. 2.3 dargestellt, an welcher sich der nachfolgende Text orientiert.

2.3.3.1.1 Outer Encoding Zu Beginn wird die zu kodierende Datei der Länge n in mehrere gleichgroße Blöcke, die SBs, eingeteilt (in der Abbildung als „Message blocks“ bezeichnet“). Somit bilden $\frac{n}{\text{Blockgröße}} = k$ SBs die Arbeitsgrundlage. Die Blockgröße kann frei gewählt werden, jedoch muss sie ausreichend klein sein, damit mindestens q Zusatzblöcke, die Auxiliary Blocks (ABs), erzeugt werden. Im Anschluss daran werden $0.55q\epsilon k$ ABs erzeugt. Hierbei sind q und ϵ frei wählbare

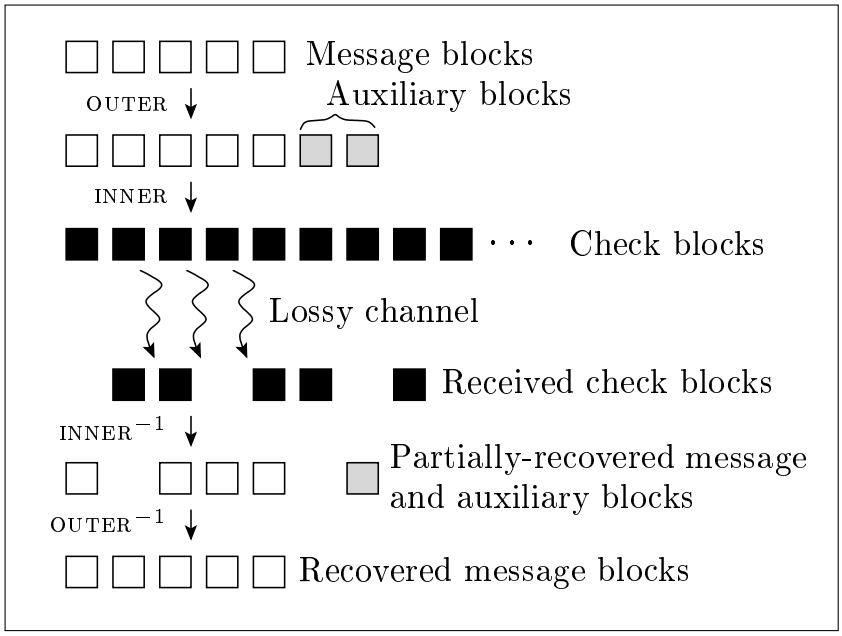

Abbildung 2.3: Allgemeiner Ablauf von Online Codes [MM03a]

Parameter. Vom Autor werden $q = 3$ und $\epsilon = 0.01$ für den praktischen Einsatz vorgeschlagen.

> Anmerkung: ϵ ist hier nicht als Reception Overhead (siehe Abschnitt 2.3) misszuverstehen. Es wird lediglich so verwendet um konsistent mit den Dokumenten [May02] und [MM03a] zu sein.

Nachdem die Anzahl der ABs festgelegt ist, muss deren Inhalt bestimmt werden. Hiermit wird mit Zuhilfenahme eines PRNG jeder SB genau q ABs zugewiesen. Somit wird klar, dass mindestens q ABs existieren müssen und eine dementsprechend kleine Blockgröße gewählt werden muss. Anzumerken ist hierbei, dass jeder SB höchstens einmal einem AB zugewiesen werden darf. Der Dateninhalt eines AB ergibt sich dann aus der **XOR**($\oplus$)-Operation über allen zugewiesenen SBs.

Die so erzeugten ABs werden an die ursprünglichen SBs angehängt. Diese zusammengefügte Nachricht bildet die Composite Message (CM), welche Ausgangspunkt für das nachfolgende Inner Encoding ist. Mit einer Datenmenge von $1 - \frac{\epsilon}{2}$ der CM lassen sich die Quelldaten mit einer Wahrscheinlichkeit von $1-(\frac{\epsilon}{2})^{q+1}$ wiederherstellen, Voraussetzung hierfür ist jedoch eine sehr große Blockanzahl (nach [MM03a]). Wieso das Outer Encoding so wichtig für das Funktionieren von Online Codes ist, wird später gezeigt. (siehe Abschnitt 2.3.3.1.3).

2.3.3.1.2 Inner Encoding Das Inner Encoding erzeugt aus der CM die zu übertragenden kodierten Blöcke, im Folgenden CBs genannt. Grundlage für die Generierung der CBs stellt die folgende **Wahrscheinlichkeitsverteilung** (nach [MM03a])

$$F = \left\lceil \frac{ln(\epsilon^2/4)}{ln(1-\epsilon/2)} \right\rceil$$
$$p_1 = 1 - \frac{1+1/F}{1+\epsilon}$$
$$p_i = \frac{(1-p_1)F}{(F-1)i(i-1)} \qquad \text{für } i = 2, \ldots, F$$

F ... maximaler CB-Grad, z.B. 2115 für $q = 3$ und $\epsilon = 0.01$
p_i ... Wahrscheinlichkeit des Grades i für einen CB

dar. In Abb. A.1 und Abb. A.2 ist die Wahrscheinlichkeitsverteilung visualisiert. Mithilfe dieser Verteilung ist es möglich CBs zu generieren. Die Generierung gliedert sich in folgende vier Schritte:

1. PRNG mit Seed initialisieren
2. Grad d mittels vorher festgelegter Wahrscheinlichkeitsverteilung „würfeln“
3. d Blöcke aus der CM mittels selbem PRNG auswählen, jedoch diesmal mit einer Gleichverteilung
   (hierbei darf jeder CM-Block maximal einmal vorkommen)
4. Daten via **XOR**($\oplus$)-Operation über zugewiesene CM-Blöcke berechnen

Für jeden CB ist es wichtig, einen nach Möglichkeit exklusiven Seed zu verwenden. Das kann beispielsweise dadurch erreicht werden, dass der Sender eine fortlaufende Nummer als Seed verwendet. Eine weitere einfache Möglichkeit wäre es, einen Zeitstempel vor der Generierung jedes CBs zu verwenden. Maymounkov macht jedoch in [May02] keine feste Vorgabe, wie diese Seedvergabe zu realisieren ist.

Die durch das Outer Encoding entstandenen CBs können nun dem Empfänger übersendet werden. Hierbei ist es lediglich nötig die eigentlichen Daten, sowie den verwendeten Seed, dem Empfänger zu schicken. Mithilfe des Seeds ist es dem Empfänger später möglich die benötigten Metadaten eines jeden CBs zu berechnen.

2.3.3.1.3 Decoding Das Decoding besteht aus dem Inner Decoding und dem Outer Decoding, jedoch lassen sich beide Vorgänge miteinander verbinden, da sie nahezu identisch sind. Der Dekodiervorgang ähnelt stark dem von LT Codes (siehe

Abschnitt 2.3.1.1), jedoch treten hierbei noch ABs auf. Das Decoding kann entweder „offline“, sprich nach Erhalt einer gewissen Menge CBs, oder „on-the-fly“ geschehen, d.h. nach Erhalt jedes CBs. In beiden Fällen wird ein CB oder AB mit Unbekanntheitsgrad 1, also mit einem unbekannten beinhalteten Block, gesucht. Dieser CB bzw. AB wird aufgelöst, womit der unbekannte Block auf den entsprechenden beinhalteten Block abgebildet wird. Hierbei kann im Falle des Auflösens eines CB entweder ein SB oder ein AB entstehen, wenn hingegen ein AB aufgelöst wird, kann daraus lediglich ein SB entstehen.

Der durch diesen Vorgang bekannt gewordene SB oder AB kann somit als bekannt markiert werden. Jetzt wird der Grad aller Blöcke um 1 verringert, die den gerade gelösten Block beinhalten. Zusätzlich müssen die Daten jedes betroffenen Blocks via **XOR**($\oplus$)-Operation mit der Datenmenge des Blocks verknüpft werden. Durch diesen Schritt können erneut Blöcke den Unbekanntheitsgrad 1 erlangen, womit der Vorgang fortgesetzt werden kann. Wenn alle SBs gelöst wurden sind, ist der Dekodiervorgang erfolgreich abgeschlossen. Besitzt der Dekodierer jedoch keine Blöcke mit einen Unbekanntheitsgrad 1 und sind noch nicht alle SBs bekannt, so schlägt das Dekodieren fehl.

Abb. 2.4 zeigt ein Beispiel eines erfolgreichen „offline“ Dekodiervorgangs. Es folgt eine Beschreibung der in jedem Schritt durchgeführten Aktionen:

1. Auflösen des linken CB → Gewinn des mittleren SB
   Entfernen aller Kanten des SB → Gradverringerung aller zugehörigen Blöcke um 1

2. Auflösen des linken CB → Gewinn des AB
   Entfernen aller Kanten des AB → Gradverringerung aller zugehörigen Blöcke um 1

3. Auflösen eines der beiden CBs → Gewinn des rechten SB
   Entfernen aller Kanten des SB → Gradverringerung aller zugehörigen Blöcke um 1

4. Auflösen des AB → Gewinn des linken CB

5. Dekodiervorgang erfolgreich abgeschlossen, da alle SBs bekannt

Bei genauer Betrachtung des Dekodiervorgangs wird klar, dass jeder SB in mindestens einem CB oder AB vorkommen muss, damit die Datei erfolgreich dekodiert werden kann. Analog und leicht abgewandelt hierzu kann man das in Kapitel 2.1

beschriebene Sammelbilderproblem sehen. Das Outer Encoding trägt durch Erstellung der ABs erheblich dazu bei, dass alle SBs vom Inner Encoder verarbeitet werden. Die Wahrscheinlichkeit, dass der Inner Encoder einen SB bei der Generierung von Blöcken über die gesamte Ursprungsdatenlänge auslässt, wird durch das Vorhandensein der ABs stark reduziert.

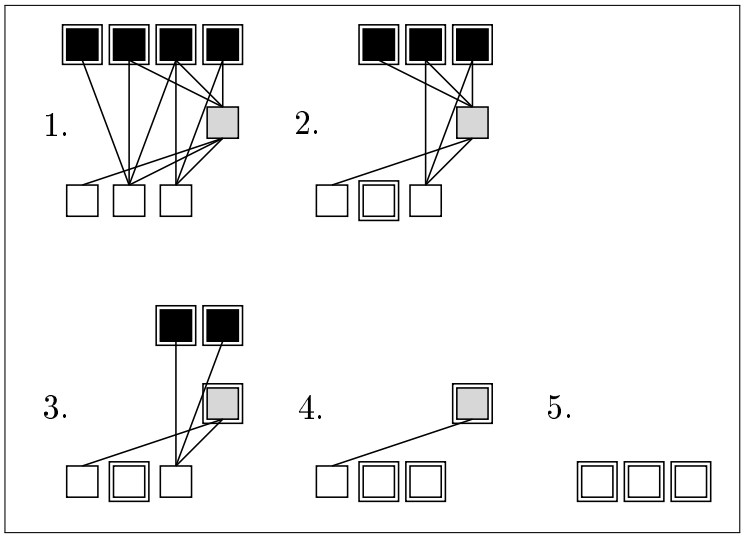

Abbildung 2.4: Dekodiervorgang von Online Codes (CB: schwarz, AB: grau, SB: weiß, doppelt umrahmt: bekannt)[MM03a]

2.4 Vergleich

	Tornado	LT	Raptor	Online
ratenlos	X	✓	✓	✓
systematisch	✓	X	✓/X	X
Kodierung pro Block	linear	leicht überlinear ($nlog(n)$)	konstant	konstant
Dekodierung pro Datei	linear	leicht überlinear ($nlog(n)$)	linear	linear
patentfrei	X	X	X	✓

Tabelle 2.1: Vergleich behandelter Kodierverfahren, Komplexität für Kodierung und Dekodierung beziehen sich hier auf die Länge der Quelldaten

Tabelle 2.1 zeigt einen direkten Vergleich von Tornado Codes und den drei behandelten ratenlosen Codes. Aufgrund der hervorragenden Eigenschaften von Online Codes, insbesondere durch die Patentfreiheit, wurden diese für die Implementierung des Prototypens gewählt.

Lediglich durch die unsystematische Verfahrensweise von Online Codes eignen sich systematische Raptor Codes für manche Anwendungsfälle besser. Ein Beispiel hierfür wäre der Download einer Datei von genau einer Quelle ohne jegliche Paketverluste. Bei diesen Raptor Codes ist durch den systematischen Aufbau keine komplexe Berechnung seitens des Dekodierers notwendig, da die ersten k kodierten Blöcke exakt den k Quellblöcken entsprechen. Es reicht hier die einzelnen Blöcke zusammenzufügen und die Datei zu speichern. Im Fall des Downloads aus mehreren Quellen oder des Downloads mit Paketverlusten verfällt dieser Vorteil jedoch wieder.

Ein weiteres Beispiel für den Nachteil des unsystematischen Kodierens ist das Stre-

aming von zeitkritischen Daten, wie beispielsweise Videos. In ihrer reinen Form sind Online Codes hierfür ungeeignet, wobei hingegen sich Raptor Codes ausgezeichnet für diesen Anwendungsfall eignen (sowohl durch systematischen Aufbau, als auch durch mögliche doppelte Unterteilung der Daten). Auf dieses Beispiel wird in Abschnitt 3.3.4.6 allerdings noch genauer eingegangen.

Abgesehen von den gerade genannten Szenarien gibt es jedoch viele Anwendungsfälle, in denen sich beide Codes hervorragend eignen.

Tornado Codes stellen keine Alternative zu Raptor und Online Codes dar, da es sich hierbei nicht um ratenlose Codes handelt. Tornado Codes wurden nur zum Vergleich hinzugezogen, damit ersichtlich ist, wie Raptor und Online Codes durch ihr voriges Outer Encoding die schlechten Eigenschaften von LT Codes überwinden.

Diese schlechten Eigenschaften von LT Codes sind der sehr hohe Kodieraufwand pro Block und das komplexere Decoding. Insbesondere bei großen Datenmengen wirken sich diese Eigenschaften stark negativ auf das Laufzeitverhalten aus, womit die Wahl auf Raptor oder Online Codes fallen musste.

3 Prototypische Implementierung von Online Codes

Im folgenden Teil der Arbeit wird auf den, während der Arbeit entstandenen, Prototypen eingegangen. Das Ziel hierbei war es, das Verfahren umzusetzen und somit zu zeigen, dass es im praktischen Einsatz wie beschrieben funktioniert. Um das zu zeigen wurde ein Prototyp entworfen, welcher eine Datei kodiert, also aus der Datei CBs generiert. Diese CBs werden an einen Dekodierer übergeben, welcher aus diesen die ursprüngliche Datei rekonstruiert.

Mithilfe des Prototypens soll es möglich sein praktische Versuche durchzuführen, womit beispielsweise der Reception Overhead in Abhängigkeit der gewählten Blockgröße untersucht werden kann. Weiterhin sollte geklärt werden, ob durch den benötigten Berechnungsaufwand des Verfahrens ein ausreichend hoher Datendurchsatz ermöglicht wird, um so Aussagen über den praktischen Nutzen von Online Codes tätigen zu können.

Gegliedert ist der nachfolgende Teil in Entwurf, Implementierung und abschließender Leistungsbewertung.

3.1 Entwurf

In den folgenden Abschnitten wird auf wichtige Aspekte während der Entwurfsphase des Prototypens eingegangen. Zum einen wird kurz abgehandelt, welche Programmiersprache aus welchen Gründen gewählt wurde. Gefolgt davon wird beschrieben, wieso großer Wert auf eine eindeutige Trennung von Kodierer und Dekodierer gelegt wurde. Im Anschluss daran werden Überlegungen zur Speicherung der ABs ausgeführt. Die Idee hinter der Aufteilung des Dekodiervorgangs sowie deren Vorteile werden nachfolgend erläutert. Abschließend wird die resultierende Architektur des Entwurfs aufgezeigt und der gesamte Kodiervorgang anhand einer schematischen Darstellung erläutert.

3.1.1 Wahl der Programmiersprache

Bei der Wahl der Programmiersprache wurde ein sehr großer Wert auf Effektivität bezogen auf den Programmieraufwand gelegt. Somit war klar, dass eine höhere Programmiersprache für die Implementierung des Prototypens gewählt wird. Der Prototyp wurde schließlich in Python umgesetzt, da diese Programmiersprache entscheidende Vorteile gegenüber anderen Programmiersprachen besitzt. Sie ist weit verbreitet, insbesondere für quelloffene Projekte (siehe [Bar13]). Ein sehr großer Umfang an mitgelieferten Bibliotheken erleichtert die Problemlösung in vielen Fällen erheblich.

Weiterhin ist Python auf vielen Plattformen lauffähig und frei verfügbar. Dadurch bedingt, dass Python eine interpretierte, dynamisch typisierte Programmiersprache ist, ist die Geschwindigkeit von Programmen meist schlechter als deren Pendants, geschrieben in Low-Level Programmiersprachen. Es hat sich jedoch gezeigt, dass sich mithilfe der sorgfältigen Wahl von Bibliotheken ein ausreichend performanter Prototyp umsetzen lässt. Auf Details zu dieser Thematik wird im Abschnitt 3.2 eingegangen.

Python wurde meinerseits bereits in vielen Projekten erfolgreich eingesetzt, womit keine zeitintensive Einarbeitungsphase vonnöten war. Dies war, neben den zuvor genannten Vorteilen, nicht zuletzt ein entscheidender Faktor für die getroffene Wahl.

3.1.2 Klare Trennung von Kodierer und Dekodierer

Für den Prototypen ist es wichtig, dass die Komponenten zum Kodieren und Dekodieren klar voneinander getrennt sind. Hiermit soll gezeigt werden, dass das Verfahren tatsächlich auf die beschriebene Weise funktioniert. Das heißt, dass zwischen Sender und Empfänger einer Nachricht initial die Dateigröße und die verwendete Blockgröße übermittelt werden müssen. Im Anschluss sollen lediglich CBs mit zugehöriger ID übertragen werden, bis der Dekodierer eine erfolgreiche Dekodierung mitteilt und somit keine weiteren CBs entgegen nimmt. Um redundanten Code zu vermeiden ist es somit nötig die Komponenten für das Outer und Inner Coding ungebunden zu entwerfen, da diese sowohl vom Kodierer, als auch vom Dekodierer benötigt werden.

Ermöglicht wird das Entkoppeln der Komponente für das Outer Coding (nachfolgend als „OuterCodingAssistant“ bezeichnet) dadurch, dass für das Berechnen der Metadaten für die AB nicht mehr als die Dateigröße und die verwendete Blockgröße benötigt wird. Da der Kodierer diese Informationen aus der Datei extrahiert

und diese an den Dekodierer übermittelt, ist es somit für beide möglich den OuterCodingAssistant zu initialisieren.

Ähnlich verhält es sich mit der Komponente für das Inner Coding (nachfolgend als „InnerCodingAssistant“ bezeichnet). Für diese Komponente wird zur Initialisierung lediglich der maximal mögliche CB-Grad benötigt. Da dieser Grad sich mit festgelegten Konstanten, der Dateigröße sowie der gewählten Blockgröße berechnen lässt, ist es auch möglich diese Komponente von Kodierer und Dekodierer loszulösen.

Weiterhin wird durch diese klare Trennung sichergestellt, dass sowohl für das Outer Coding, als auch für das Inner Coding stets gleiche Werte seitens des Kodierers und Dekodierers berechnet werden. Dieser Fakt verringert die Anfälligkeit des Prototypens auf Programmierfehler erheblich.

3.1.3 Speicherung der Auxiliary Blocks

Nach Erstellung der ABs ist es sinnvoll, diese auf einem Massenspeicher abzulegen, damit sie nicht unnötigen Platz im Arbeitsspeicher beanspruchen. Für das Speichern der Daten kamen folgende Möglichkeiten in Frage:

I Anhängen an die Quelldatei

II Speichern als Zusatzdatei auf Basis des Dateinamens

III Speichern als Zusatzdatei auf Basis einer Prüfsumme

Option I schied als praktikable Lösung aus, da hier eine Modifikation der Quelldatei vorgenommen wird, was in jedem Fall zu vermeiden ist. Durch unvorhergesehenes Fehlverhalten kann es hierbei zu vollständigem Datenverlust kommen, da die zu kodierende Datei mit Schreibberechtigungen geöffnet werden müsste.

Somit war relativ schnell klar, dass die Daten der ABs separat abgelegt werden müssen. Hierbei besteht das Problem der Zuordnung der Zusatzdaten zu den entsprechenden Quelldaten. Eine Möglichkeit bildet Option II, wobei die Zuordnung der Zusatzdaten zu den Quelldaten hier problematisch werden kann, wenn eine Datei umbenannt oder verschoben wird. Weiterhin müsste man die Datei explizit auf Veränderungen prüfen, sowohl in Bezug auf Größe, als auch auf Inhalt.

Letztendlich stellt Option III die gewählte Lösung dar, da hier eben genannte Probleme von Variante II nicht auftreten können.

> Anmerkung: Der Fall einer Kollision bei Verwendung einer Prüfsumme wird hier nicht gesondert betrachtet, da bei Wahl eines guten Verfahrens Kollisionen sehr selten auftreten. Für den Fall müsste man einen Mechanismus zur Kollisionserkennung integrieren, was jedoch beim Prototypen nicht umgesetzt wurde.

Die Datei wird durch die Prüfsummenberechnung implizit auf Veränderungen untersucht. Weiterhin bereitet somit das Verschieben oder das Umbenennen einer Datei keinerlei Schwierigkeiten mehr, da die Prüfsumme lediglich vom Inhalt einer Datei abhängig ist.

3.1.4 Aufteilung der Dekodierung

Während des Dekodiervorgangs muss, bedingt durch den Aufbau von Online Codes, viel Prozessorzeit beansprucht und eine große Menge an Massenspeicherzugriffen durchgeführt werden. Um die Massenspeicherzugriffe während des Empfangens einer Datei möglichst niedrig zu halten, wurde der Dekodiervorgang in zwei Teile aufgesplittet.

Der erste Teil geschieht während des Empfangens von CBs und dient der **Lösungsfindung**. Es wird hierbei lediglich mit den Metadaten der CBs gearbeitet, die eigentlichen Daten werden jedoch nicht angetastet. Konkret heißt das, dass ein Plan zur vollständigen Dekodierung erarbeitet wird. Dieser Plan beinhaltet die Abarbeitungsreihenfolge der einzelnen CBs, welche zu einer erfolgreichen Dekodierung führen (vorausgesetzt, dass ausreichend CBs zur Verfügung stehen).

> Anmerkung: Ein einfaches Abspeichern von CBs ohne jegliche Lösungsberechnung ist nicht empfehlenswert, da in dem Fall der Empfänger einer Datei dem Sender nicht signalisieren kann, wann er ausreichend CBs empfangen hat. Er könnte lediglich einen gewissen Overhead pauschal einplanen, was jedoch dazu führen würde, dass in den meisten Fällen mehr Datenverkehr stattfinden müsste, als eigentlich nötig.

Erst wenn dieser Plan erfolgreich erstellt wurde, also ausreichend CBs dem Dekodierer übergeben wurden, kann das eigentliche Dekodieren der Daten erfolgen.

Somit findet erst im zweiten Teil des Dekodiervorgangs das **Auflösen** statt. Hierbei wird der im ersten Teil erstellte Plan verwendet, um die CBs in richtiger Reihenfolge miteinander zu verknüpfen und so die SBs zu rekonstruieren. Dieser Prozess ist mit der großen Anzahl an **XOR**($\oplus$)-Operationen rechenaufwändig und bringt außerdem eine hohe Zugriffsrate auf den Massenspeicher mit sich.

Ein weiterer Vorteil der getrennten Berechnung der Lösung und der Nutzdaten ist die Tatsache, dass die Nutzdaten erst dann berechnet werden, wenn die Datei auch tatsächlich dekodiert werden kann. Gesetzt den Fall, dass eine Quelle wider Erwarten nicht ausreichend CBs liefern kann, z.B. durch den Ausfall der Quelle, dann wurde lediglich ein kleiner Teil des Berechnungsaufwand fruchtlos aufgebracht. Im Praxisbeispiel einer Peer-To-Peer Tauschbörse, welche üblicherweise im Hintergrund läuft, während der Nutzer den Computer anderweitig verwendet, würde somit ein Minimum an Rechenaufwand während des Transfers durchgeführt werden. Der rechenaufwändige Prozess des Dekodierens beeinträchtigt den Nutzer somit erst nach dem vollständigen Transfer für eine kurze Zeit.

3.1.5 Gesamtvorgang

Die entstandene Architektur ist in Abb. 3.1 dargestellt. Zwei weitere Abb. zeigen die jeweils relevanten Teile für Kodierung (siehe Abb. A.3) und Dekodierung (siehe Abb. A.4) mit dem entsprechenden anderen Teil ausgegraut. In den Abb. ist die in Abschnitt 3.1.2 beschriebene Trennung von Kodierer und Dekodierer sichtbar. Weiterhin wird auch die in dem Abschnitt beschriebene Ausgliederung der Komponenten für Inner Coding und Outer Coding ersichtlich.

> Anmerkung: Die Abbildung 3.1, samt zugehörigem Text, befindet sich auf der nachfolgenden Doppelseite, da diese nebeneinander leichter zu verstehen sein sollten.

3.1.5.1 Encoding

Zu Beginn wird der Encoder mit einem Dateinamen, einer Blockgröße und optional mit einem Seed initialisiert. Es folgt eine SHA-1-Prüfsummenberechnung für das spätere Speichern der ABs. Im Anschluss daran wird der OuterCodingAssistant mit der Dateigröße und der Blockgröße initialisiert, welcher im Gegenzug die Mappings für die Erstellung der ABs liefert. Mithilfe dieser sogenannten „auxMaps“ und der SBs ist es möglich die ABs zu erstellen und auf dem Massenspeicher abzulegen. Somit ist das Outer Encoding abgeschlossen, welches die Grundlage für das Inner Encoding darstellt.

Nachfolgend können mit der Methode „GetCheckBlock“ CBs generiert werden. Hierfür wird dem „InnercodingAssistant“ (welcher mit der Blockanzahl der CM initialisiert wird) eine CB-ID übergeben, worauf dieser den Grad und die beinhaltenden Blockindizes für den CB zurückliefert. Der Encoder verknüpft jetzt alle angegebenen Blöcke mit der **XOR**($\oplus$)-Operation und gibt die resultierenden Daten inklusive CB-ID aus. Somit kann ein Strom an CBs erzeugt werden, welche dem nachfolgend beschriebenen Decoder als Eingabe dienen.

3.1.5.2 Decoding

Neben einer Menge an CBs benötigt der Decoder initial noch folgende Metadaten: Dateigröße und verwendete Blockgröße.

> Anmerkung: Die Verwendung eines vom Standardwert abweichenden Seeds für das Outer Encoding ist der Einfachheit halber nicht mit berücksichtigt. In diesem Fall müsste der Encoder dem Decoder den verwendeten Wert mitteilen, ansonsten würde es zu Fehlern bei der Dekodierung kommen.

Der Decoder initialisiert, genau wie der Encoder, den „OuterCodingAssistant“, womit dieser das gleiche Verhalten wie beim Encoding aufweist. Im Anschluss daran kann jetzt jeder CB dem Decoder übergeben werden. Dieser kann mithilfe des „InnerCodingAssistants“ und der CB-ID die Metadaten des entsprechenden CBs rekonstruieren und den im Abschnitt 3.1.4 beschriebenen Prozess des Dekodierens anstoßen. Nachdem ausreichend CBs empfangen wurden ist es dem Decoder möglich die SBs zu rekonstruieren und diese auf einem Massenspeicher abzulegen.

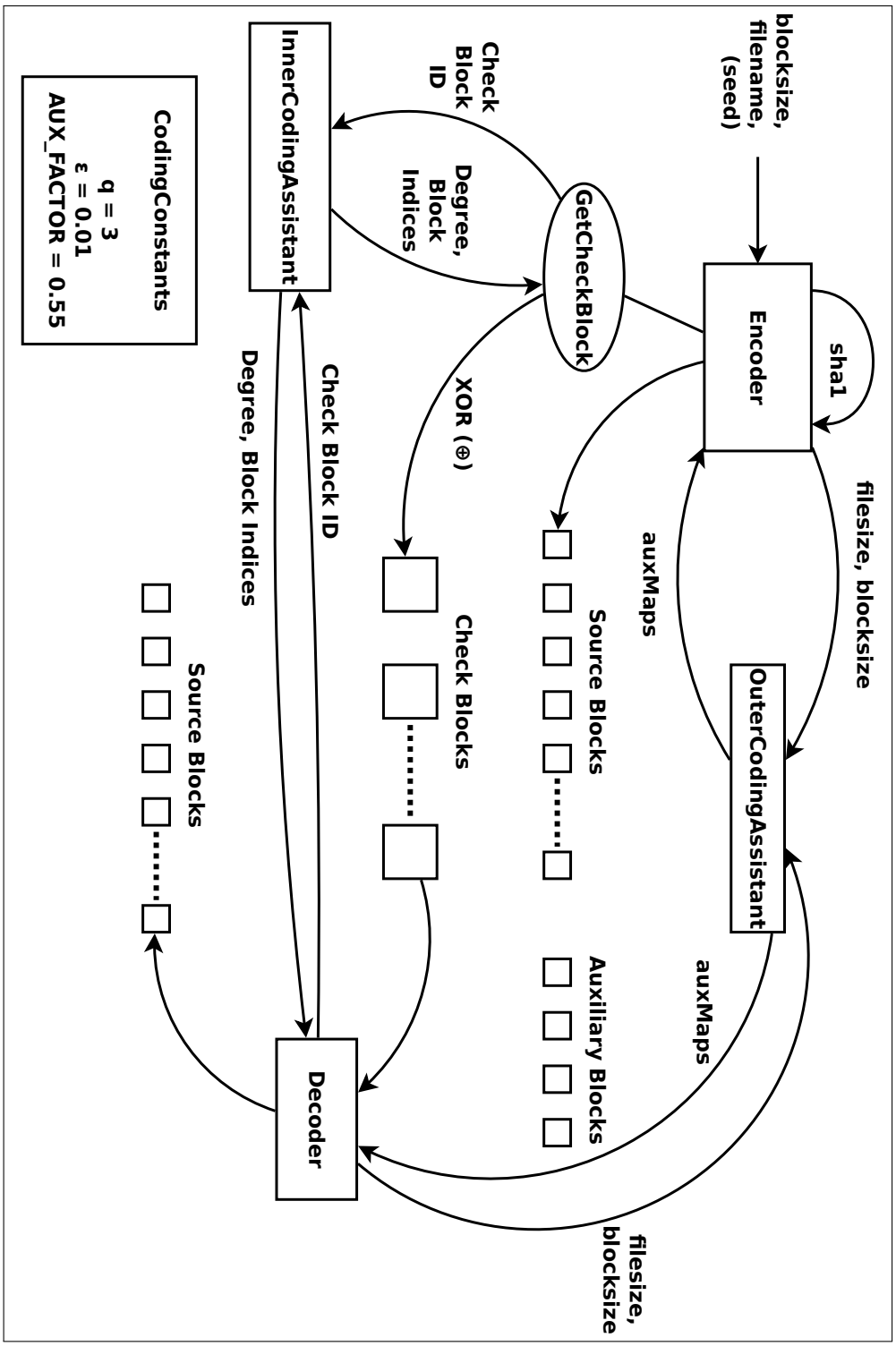

Abbildung 3.1: Architektur des Prototypens

3.2 Implementierung

Gegenstand dieses Abschnittes ist die Behandlung einiger Kernkomponenten des Prototypens, sowie die Erläuterung der Umsetzung komplexerer Sachverhalte und Vorgänge. Insgesamt soll nur ein grober Überblick gegeben werden, sodass nicht auf alle Details der Implementierung eingangen wird. Die Quellen des Prototypens sind frei zugänglich, sodass - bei Bedarf - Details in ihr nachverfolgt werden können.

Die Implementierung wurde auf einem Lenovo Thinkpad X200 entwickelt, es folgt eine Auflistung der relevanten Systeminformationen:

CPU Intel® Core™2 Duo P8600 (3M Cache, 2.40 GHz, 1066 MHz FSB)

RAM 2x 4GB Samsung DDR3-1333 PC3-10600 CL9 SO-DIMM

SSD Intel® SSD 320 Series (160GB, 2.5in SATA 3Gb/s, 25nm, MLC)

OS Ubuntu 12.04.3 LTS x86_64

Kernel 3.5.0-43-generic

Python python2.7_2.7.3-0ubuntu3_amd64

numPy python-numpy_1.6.1-6ubuntu1_amd64

Die nachfolgenden Laufzeiten und Performance-Angaben beziehen sich auf die Ausführung des Prototypens mit den aufgelisteten Komponenten. Je nach verwendetem System können die Laufzeiten von den genannten Abgaben abweichen.

3.2.1 Zufallszahlengenerator

Als Zufallszahlengenerator wurde der in Python 2.7 mitgelieferte Mersenne-Twister gewählt. Dieser PRNG hat eine sehr große Periode von $2^{19937} - 1 \approx 4,3 * 10^{6001}$ (siehe [Mata]). Das heißt, dass sich die Zufallszahlen erst nach dem $\approx 4,3 * 10^{6001}$-maligen „Würfeln“ wiederholen würden. Dadurch wird sichergestellt, dass selbst bei Wahl eines festen Seeds für einen CB-Stream (siehe 3.2.7) in der Praxis eine quasi unendliche Menge an CBs generiert werden kann.

Wäre diese Periode sehr kurz, so bestünde die Gefahr, dass nach einer relativ kleinen Menge von generierten CBs erneut die selben CBs generiert werden. Das rührt daher, dass deren Metadaten nach überschreiten der Periode auf bereits verwendeten Zufallszahlen basieren.

Der Mersenne-Twister ist nicht für kryptographische Anwendungen geeignet, was

jedoch für die Verwendung im Prototypen kein Problem darstellt. Viel mehr macht diese Eigenschaft den PRNG überhaupt erst brauchbar für den gegebenen Anwendungszweck, da dieser für das Funktionieren des Verfahrens sich in jedem Fall deterministisch verhalten soll. Das heißt, dass die ausgegeben Reihenfolge an Zufallszahlen lediglich vom verwendeten Seed abhängig ist. Werden also mehrere PRNG mit dem selben Seed initialisiert, so **müssen** diese exakt die selben Zufallszahlen liefern.

Der Mersenne-Twister besitzt, durch die Arbeitsweise bedingt, die Eigenschaft, dass er relativ viel Arbeitsspeicher benötigt (arbeitet auf 624 Wörtern, davon jedes 4 Byte groß, siehe [Matb]). Im Fall des Prototypens ist diese Eigenschaft nicht relevant, da auf einem modernen Computer ausreichend Arbeitsspeicher zur Verfügung steht. Beim Einsatz des Verfahrens auf Hardware mit beschränkten Ressourcen ist es eventuell nötig diesen PRNG durch einen anderen zu ersetzen. Welche praktische Relevanz der Einsatz von Online Codes auf z.B. eingebetteten Systemen darstellt wird in Abschnitt 3.3.4 erläutert. Bei der Implementierung wurde stets darauf geachtet, dass sich der PRNG nachträglich ohne viel Aufwand austauschen lässt.

> Anmerkung: Die Kompatibilität von Kodierer und Dekodierer ist nur beim Einsatz des gleichen PRNG auf beiden Seiten gegeben.

Ein weiterer Vorteil ist die weite Verbreitung des Mersenne-Twisters. Es lassen sich im Internet für eine sehr große Anzahl an Programmiersprachen bereits Implementierungen auffinden, darunter z.B. C, C++, C#, Go, Java, Perl und viele weitere mehr. Das ermöglicht die Programmierung von Kodierern und/oder Dekodierern, welche (durch sorgfältige Initialisierung des PRNG) Kompatibilität zu diesem Prototypen besitzen, ohne den PRNG von Grund auf implementieren zu müssen.

3.2.2 Prüfsummenbildung

Wie in Abschnitt 3.1.3 bereits erläutert, wird zum Speichern der ABs auf einem Massenspeicher eine Prüfsumme benötigt. Es wurde sich für das bekannte und weit verbreitete SHA-1-Verfahren entschieden. Dieses arbeitet ausreichend schnell, es benötigt lediglich ∼4s für eine 800MB-Datei, welche zufällige Daten beinhaltet. Diese benötigte Rechenzeit ist gegenüber der beanspruchten Zeit für Kodierung und Übertragung einer 800MB-Datei vernachlässigbar. Weiterhin wird durch diese investierte Rechenzeit ein erheblicher Komfortgewinn erzielt, welcher bereits in Abschnitt 3.1.3 erwähnt wurde.

> Anmerkung: Es sind bereits mehrere Angriffstechniken auf SHA-1 bekannt, diese ändern jedoch nichts an der Tauglichkeit des Verfahrens.

Es liegt hier, wie bereits bei der Wahl des PRNG, kein kryptographischer Anwendungsfall vor.

```
import hashlib
from functools import partial

def sha1sum(fd):
    fpos = fd.tell()
    fd.seek(0)
    d = hashlib.sha1()
    for buf in iter(partial(fd.read, 8192), b''):
        d.update(buf)
    fd.seek(fpos)
    return d.hexdigest()
```

Abbildung 3.2: Funktion zur Generierung der SHA-1 Prüfsumme einer Datei

Abb. 3.2 zeigt die Funktion zur Prüfsummengenerierung einer Datei. Es wird mit dem hashlib-Modul, welches in der Python Standard Library enthalten ist, gearbeitet. Mithilfe dieses Moduls ist es in 3 Schritten möglich, eine Prüfsumme einer Datei zu erstellen. Diese sind: Initialisierung eines SHA-1-Objekts mittels `hashlib.sha1()`, im Anschluss Dateneingabe mittels `update(buf)`, gefolgt von einem `hexdigest()` um die Prüfsumme in hexadezimaler Schreibweise zu erhalten.

Der `update`-Vorgang wird hierbei bewusst blockweise durchgeführt, da ansonsten die komplette Datei zur Prüfsummengenerierung in den Arbeitsspeicher geladen werden müsste. Auf dem Testsystem hat sich gezeigt, dass die Berechnung mit 8kB großen Blöcken am wenigsten Zeit in Anspruch nimmt. Die Berechnung mittels dieser Funktion ist schneller als mit dem Kommandozeilenprogramm `sha1sum` aus dem Debian-Paket `GNU coreutils 8.12.197-032bb`. Die Geschwindigkeit der Python-Implementierung rührt daher, dass das `hashlib`-Modul auf die OpenSSL-Bibiliothek von Linux zurückgreift. Somit wird sichergestellt, dass der sehr rechenlastige Prozess der Prüfsummenberechnung nicht vollständig interpretiert werden muss, wie es eigentlich bei Python-Code üblich ist.

3.2.3 XOR(⊕)-Realisierung

Da die **XOR**(⊕)-Verknüpfung ein wesentlicher Bestandteil von Online Codes ist, musste hierfür eine ausreichend effiziente Lösung gefunden werden. Python bietet standardmäßig nur einen bitweisen **XOR**(⊕)-Operator für Ganzzahlen an. Es ist jedoch nötig ganze Datenblöcke miteinander zu verknüpfen, welche in Python

durch Strings dargestellt werden. Eine Berechnung der Verknüpfung von Strings über den Umweg der Aufteilung dieser und Umwandlung der Teile in Ganzzahlen war nicht zielführend. Dadurch bedingt, dass Python-Code interpretiert wird, war diese Lösung sehr langsam und für große Datenmengen nicht praktikabel.

```
from numpy import bitwise_xor, fromstring, dtype

def xor(a,b):
    for i in (8,4,2,1):
        if not len(a) % i: break
    if i == 8: dt = dtype('<Q8');
    elif i == 4: dt = dtype('<L4');
    elif i == 2: dt = dtype('<H2');
    else: dt = dtype('B');
    return bitwise_xor(fromstring(a, dtype=dt),
                       fromstring(b, dtype=dt)).tostring()
```

Abbildung 3.3: Funktion zur XOR(⊕)-Verknüpfung zweier Strings

Eine bessere Alternative stellt hier die Funktion `bitwise_xor` von der Python-Erweiterung `NumPy` dar. Abb. 3.3 zeigt die implementierte Python-Funktion zur XOR(⊕)-Verknüpfung zweier Strings. Die Strings werden hierbei als Array größtmöglicher Ganzzahlen interpretiert, also je nach Länge des Strings als $8, 4, 2$ oder 1 Byte Integer. Ein Versuch hat ergeben, dass diese Implementierung bereits mindestens um den Faktor 50 schneller ist als die o.g. einfache Python-Variante.

Durch das Einbinden von C/C++ Quellcode in Python ist weiteres Verbesserungspotenzial vorhanden, jedoch würde dadurch die Plattformunabhängigkeit leiden. Dieses Potenzial wurde allerdings nicht voll ausgeschöpft, da die erzielte Geschwindigkeit für den Prototypen ausreichend ist. Es wurde beim Prototypen darauf geachtet, dass die Funktion einfach ersetzt werden kann, um somit bei Bedarf später Optimierungen vorzunehmen.

3.2.4 Ermitteln des Blockgrades

Für die Ermittlung der Blockgrade liegt die Wahrscheinlichkeitsverteilung aus Abschnitt 2.3.3.1.2 zugrunde. Diese wird mithilfe der Funktion aus Abb. 3.4 erstellt. Anzumerken ist hierbei, dass F entgegen der mathematischen Gleichung das Minimum von dem berechneten F und der Blockanzahl (Anzahl SBs + Anzahl ABs) der CM ist. Die Bildung dieses Minimums übernimmt die Methode `GetMaxDegree()`. Da diese reine Wahrscheinlichkeitsverteilung nicht sehr praktikabel für die Bestimmung des Grades für einen CB ist, wird diese noch umgewandelt. Die in Abb. 3.5

```
def __GetDistribution(self):
    F = float(self.GetMaxDegree())
    dist = [self.__P_1(F)]
    dist.extend((self.__P_i(i, F) for i in xrange(2, int(F+1))))
    return dist
```

Abbildung 3.4: Funktion zur Berechnung der Wahrscheinlichkeitsverteilung von CB-Graden

```
def __GetAccumulatedDistribution(self):
    dist = self.__GetDistribution()
    for i, dummy in enumerate(dist):
        if i > 0: dist[i] = dist[i] + dist[i-1]
    return dist
```

Abbildung 3.5: Konvertierung der Wahrscheinlichkeitsverteilung

dargestellte Funktion summiert die einzelnen Wahrscheinlichkeiten der jeweiligen Vorgänger auf, sodass gilt:

$$p_{i-1} < p_i < p_{i+1} \quad \text{für } i = 2, \ldots, F-1$$

Diese aufsummierten Wahrscheinlichkeiten der Form $(p_1, p_1+p_2, \ldots, p_1+p_2+\ldots+p_{F-1}, 1.0)$ werden unter `self.__accDist` abgespeichert, da diese für die Generierung eines jeden CBs benötigt werden. Mithilfe von der in Abb. 3.6 dargestellten

```
def __GetCheckBlockDegree(self):
    x = self.__rand.random()
    for deg, border in enumerate(self.__accDist):
        if x > border:
            continue
        elif border < x < self.__accDist[deg+1]:
            return deg+2
        elif x < border:
            return deg+1
    raise RuntimeError("shouldn't be here")
```

Abbildung 3.6: Funktion zur Bestimmung von CB-Graden

Funktion lässt sich der Grad eines neuen CBs bestimmen. Es wird zuerst eine Zufallszahl `x` zwischen 0 und 1 „gewürfelt“ und im Anschluss danach geprüft, zwischen welchen beiden Werten des Arrays der gewürfelte Wert `x` liegt. Hierbei stellt

in der Schleife `border` die aktuelle obere Grenze aus dem Array und `deg` den aktuellen Grad dar. Dem Grad muss, je nach entsprechendem Fall, 1 bzw. 2 vor der Rückgabe addiert werden, da bei 0 angefangen wird zu zählen, Grade jedoch erst ab dem Wert 1 Sinn ergeben.

3.2.5 Blockmappings für Outer Encoding

Beim Outer Encoding muss ein SB genau `Q` (Prototyp: Q=3) verschiedenen ABs zugewiesen werden. Abb. 3.7 zeigt, wie diese Zuordnung umgesetzt ist. Es wird iterativ für jeden SB-Index eine Zuweisung zu genau `Q` ABs vorgenommen. Nach Abschluss des Vorgangs wird eine Liste, wie in der Funktionsdokumentation beschrieben, zurückgegeben. Hierbei lässt sich erkennen, dass der Vorgang umso länger dauert, desto weniger ABs vorhanden sind. Im schlimmsten Fall sind lediglich Q ABs vorhanden, was darin resultiert, dass jeder AB-Index mindestens einmal als `rint` gewürfelt werden muss. In der Praxis sollte dieser Fall jedoch selten auftreten, da durch eine sorgfältig gewählte Blockgröße die Anzahl der ABs ein Vielfaches von Q darstellt. In diesem Fall ist die Wahrscheinlichkeit für ein mehrfaches Würfeln des gleichen Werts relativ gering und muss somit nicht gesondert behandelt werden.

```
def GetEncodingPerAuxiliaryBlock(self):
  """
  returns list of
  [srcIndex0, srcIndex1, ..., srcIndexN] which are in auxblock 0
  [srcIndex0, srcIndex1, ..., srcIndexN] which are in auxblock 1
  ...
  [srcIndex0, srcIndex1, ..., srcIndexN] which are in auxblock N
  """
  self.__ResetState()
  auxMaps = [[] for dummy in xrange(self.__auxBlockcount)]
  for blockindex in xrange(self.__blockcount):
   for dummy in xrange(CodingConstants.Q):
     while True:
       rint = self.__rand.randrange(0, self.__auxBlockcount)
       if blockindex not in auxMaps[rint]:
         break #and jump in the next line
     auxMaps[rint].append(blockindex)
  return auxMaps
```

Abbildung 3.7: Funktion zur Berechnung der Mappings für das Outer Encoding

3.2.6 Blockmappings für Inner Encoding

Ähnlich der Zuweisung von Blöcken im vorigen Abschnitt funktioniert die Zuordnung von Blöcken beim Inner Encoding. Da hier jedoch die CM als Ausgangspunkt dient, müssen sowohl SBs als auch ABs einem Block zugewiesen werden, welcher dann einen CB darstellt. Die Anzahl der zu verknüpfenden Blöcke ist jedoch nicht konstant, sondern wird vorher zufällig bestimmt, wie in Abschnitt 3.2.4 erläutert wurde. Abb. 3.8 zeigt die Funktion, welche einen zufällig bestimmten Grad ent-

```
def __GetBlocks(self, deg):
  blockIndices = []
  for dummy in xrange(deg):
    while True:
      rint = self.__rand.randrange(0, self.__maxDegree)
      if rint not in blockIndices: break
    blockIndices.append(rint)
  return blockIndices
```

Abbildung 3.8: Funktion zur Generierung einer Indizeliste für das Inner Encoding

gegennimmt und eine Liste von zufälligen Blockindizes zurückliefert. Die Blöcke werden fortlaufend nummeriert, an die Menge der SBs werden die ABs angehängt, somit ist `self.__maxDegree` die gesamte Anzahl von SBs und ABs.

3.2.7 Block-Klassen

Ein essenzieller Bestandteil des Prototypens sind die Klassen für die verschiedenen Arten von Blöcken. Hierbei wurden lediglich Klassen für komplexere Blöcke umgesetzt, welche CBs und ABs sind. SBs besitzen außer einem Blockindex keine Metadaten, weshalb hierfür keine extra Klasse entworfen wurde. Die Initialisierungsmethode der Basisklasse ist in Abb. 3.9 dargestellt. Diese Basisklasse ent-

```
class Block(object):
  def __init__(self, streamID, number, data, blocks=None):
    self.__streamID = streamID
    self.__number = number
    self._data = data
    self.__blocks = blocks
    self.__known = []
```

Abbildung 3.9: Initialisierungsmethode der Block-Klasse

spricht exakt der Implementierung der CB-Klasse, jedoch erbt auch die AB-Klasse von ihr, sodass eine direkte Benennung der Basisklasse als CB-Klasse irreführend wäre. Bei der Initialisierung kann eine Kombination aus `streamID` und `number` verwendet werden, womit dem Decoder mitgeteilt wird, dass dieser Block einem gewissen Blockstrom zugehört. Mit diesem Wissen muss dieser die Metadaten des Blocks anders bestimmen, als wenn lediglich eine `streamID`, in dem Fall dann fungierend als CB-ID, angegeben ist. Die Gedanken hinter diesem Ansatz werden in Abschnitt 3.3.4 näher erläutert.

In dem Attribut `data` werden die Rohdaten eines Blocks gespeichert, `blocks` wird genutzt, um die Liste der beinhaltenden Blockindizes zu speichern. Das Attribut `known` ist eine Liste, welcher während des Dekodiervorgangs bereits gelöste Blockindizes hinzugefügt werden. Mithilfe von `blocks` und `known` kann somit der Unbekanntheitsgrad eines Blocks bestimmt werden, welcher eine wichtige Eigenschaft für den Dekodiervorgang darstellt.

In Abb. 3.10 ist sowohl die Initialisierungsmethode, als auch die Methode zur Rückgabe der Rohdaten eines ABs dargestellt. Als weiteres Attribut besitzt diese

```
class Auxblock(Block):
  def __init__(self, cb, blocks=None):
    Block.__init__(self, None, None,
                   "\x00"*len(cb.GetData()), blocks)
    self.__cb = cb

  def GetData(self):
    return xor(self.__cb.GetData(), self._data)
```

Abbildung 3.10: Details der Auxblock-Klasse

Klasse eine Referenz auf einen CB namens `cb`, womit der Dekodierer einem AB die Eigenschaft vergeben kann, aus welchem CB er extrahiert wurde. Das ist für den späteren Dekodiervorgang sehr wichtig, da hierfür die Rohdaten des CBs benötigt werden.

Bei Initialisierung können die reinen Rohdaten noch nicht übergeben werden, da diese zum Zeitpunkt der **Lösungsfindung** (siehe Abschnitt 3.1.4) noch nicht berechnet wurden. Somit müssen die benötigten Rohdaten erst zum Zeitpunkt des **Auflösens** (siehe Abschnitt 3.1.4) von dem entsprechenden CB bezogen werden.

3.2.8 Decoding

Die komplette Funktionsweise des Decoders bis in das kleinste Detail zu erläutern, würde den Rahmen sprengen, daher wird hier lediglich auf relevante Teile eingegangen.

Der Decoder bekommt die Dateigröße und die gewählte Blockgröße bei der Initialisierung übergeben. Hiermit errechnet er die Anzahl der SBs und ABs und kann somit die Hilfsobjekte für das Inner- und Outer Coding initialisieren, mit denen Metadaten von CBs und ABs rekonstruiert werden können. Weiterhin hält er eine Liste von Listen im Speicher, worin die CBs und ABs, sortiert nach Unbekanntheitsgrad, aufgelistet werden. Besonders wichtig ist hierbei die Liste, welche sich am Index 1 befindet, da hier die CBs eingeordnet werden, welche auf SBs bzw. ABs abgebildet werden können. Da ABs vom Decoder wie CBs behandelt werden, können hier ebenso ABs während des Dekodiervorgangs einsortiert werden. Mit Ablauf des Dekodiervorgangs wird durch das Auflösen von CBs erreicht, dass immer mehr CBs im Grad verringert und somit in Listen weiter vorn eingegliedert werden.

Diese Liste allein reicht allerdings nicht aus, um das Dekodieren effizient durchzuführen. Deswegen hält der Dekodierer noch eine weitere Liste von Listen im Speicher. In dieser kann mit einem beliebigen SB- bzw. AB-Index auf eine Liste aller CBs zugegriffen werden, welche den jeweiligen SB bzw. AB beinhalten. Diese ist insofern extrem wichtig, da nach erfolgreicher Wiederherstellung eines SB bzw. AB sofort alle betroffenen CBs um den Grad 1 verringert werden können. Eine Implementierung ohne diese Liste wäre ebenfalls möglich, jedoch müsste dann die Liste aller CBs durchgegangen werden um CBs auszuwählen, welche den entsprechenden Block beinhalten. Dieser Vorgang würde sehr viel Rechenzeit verschwenden, weshalb sich dafür entschieden wurde die Liste dauerhaft im Speicher zu halten.

Zusätzlich zu den genannten Daten wird eine Liste aller bereits aufgelösten SBs und ABs gepflegt. Diese wird beim Hinzufügen von neuen CBs benötigt, da diese ggf. bereits gelöste Blöcke beinhalten und somit deren Metadaten verändert werden müssen. Wählt man aus der Liste alle SBs aus, so kann mit deren Anzahl, sowie der gesamten Anzahl an SBs der kodierten Datei, eine Aussage über den Dekodierfortschritt gegeben werden.

Die Ergebnisse der Lösungsfindung werden in einer weiteren Liste gespeichert. Diese Liste ist eine Aneinanderreihung von CBs, deren Reihenfolge extrem wichtig ist. Nachdem alle SBs als gelöst markiert wurden, muss der Dekodierer diese CBs

in exakt dieser Reihenfolge nacheinander abarbeiten, um die SBs rekonstruieren zu können.

Diese Datenverwaltung geschieht vollständig im Hintergrund, nach der Initialisierung des Dekodierers sind von außen lediglich folgende Methoden relevant:

AddBlock(cb) fügt einen neuen CB dem Decoder hinzu, stößt je nach Vorhandensein von neuen Grad-1 Blöcken den Vorgang zur Lösungsfindung an. Falls die Datei bereits vollständig gelöst wurde, kehrt die Methode direkt zurück.

Decode() nutzt die gefundene Lösung um alle SBs zu rekonstruieren. Falls noch nicht alle Quellblöcke gelöst wurden, kehrt diese Methode sofort zurück.

WriteTo(path) speichert alle SBs aneinandergereiht ab, sodass diese die Quelldatei repräsentieren. Wenn noch nicht alle SBs rekonstruiert sind, kehrt diese Methode sofort zurück.

Eine Integration von Online Codes auf Basis des Prototypens in eine andere Software stellt somit keinen großen Einarbeitungsaufwand für andere Entwickler dar. Für welche Anwendungsfälle sich der Einsatz lohnt, wird in Abschnitt 3.3.4 geklärt.

Auf Basis der entstandenen Implementierung wird nachfolgend eine Leistungsbewertung durchgeführt.

3.3 Leistungsbewertung

Im diesem Abschnitt wird eine Leistungsbewertung des entstandenen Prototypens vorgenommen. Hierbei erfolgt ebenso eine Bewertung des praktischen Einsatzes von Online Codes, da diese mithilfe einer funktionierenden Implementierung leichter durch Messwerte zu belegen ist. Die ermittelten Werte basieren auf dem Einsatz des Prototypens auf dem System, welches im Abschnitt 3.2 beschrieben wurde.

3.3.1 Implementierungsaufwand

Der Implementierungsaufwand eines Verfahrens spielt für dessen praktischen Einsatz meistens eine entscheidende Rolle. Mal angenommen, dass Online Codes in einer Tauschbörsen-Software Anwendung finden sollen. Hierbei wäre es von Vorteil, wenn das Kodierverfahren, welches lediglich einen kleinen Teil der gesamten Software darstellt, relativ leicht und schnell zu implementieren ist. Würde der Implementierungsaufwand sehr groß sein, so könnte das ein Grund für die Wahl eines anderen, möglicherweise nicht so leistungsfähigen, Verfahrens sein.

Wie am entstandenen Prototypen sichtbar ist, bieten Online Codes diesen Vorteil des recht geringen Implementierungsaufwands. Das Verfahren wird im Dokument „Rateless Codes and Big Downloads“ (siehe [MM03a]) sehr verständlich und lückenlos auf wenigen Seiten erläutert, sodass bei einer Implementierung gemäß den Vorgaben keine Unklarheiten oder Probleme auftreten sollten. Bei Wahl eines geeigneten PRNG ist auch die Interoperabilität zwischen verschiedenen Plattformen implizit gegeben, sodass bezüglich Implementierungsaufwand nichts gegen den Einsatz von Online Codes spricht.

3.3.2 Zusammenhang Reception Overhead/Laufzeit

In Tabelle 3.1 sind Ergebnisse des Kodiervorgangs unter Verwendung verschiedener Blockgrößen dargestellt. Eine Darstellung der Reception Overhead-Messwerte ist in den Abb. A.5 bis A.15 zu finden. Es werden je verwendeter Blockgröße minimaler, maximaler und durchschnittlicher Reception Overhead, sowie eine durchschnittliche Laufzeit aufgeführt. Wie bereits in der theoretischen Betrachtung erwähnt, sollte es möglich sein, den benötigten Reception Overhead durch höheren Berechnungsaufwand zu verringern. Das kann durch die Funktionsweise von Online Codes jedoch nur in gewissen Grenzen erfolgen, eine Verringerung des Overheads auf nahezu Null ist hierbei nicht möglich.

Diese gegenläufige Abhängigkeit von Reception Overhead und Berechnungsaufwand wird in der Tabelle sichtbar. Je kleiner die gewählte Blockgröße ist, aus

Blockgröße	SB-Anzahl	Reception Overhead Minimum	Mittelwert	Maximum	∅-Laufzeit in s
512kB	160	0.0563	0.7719	1.4938	0.021 (+1.9)
256kB	320	0.05	0.2907	1.4969	0.047 (+1.5)
128kB	640	0.05	0.1451	0.7438	0.125 (+0.8)
64kB	1280	0.0422	0.0920	0.3602	0.373 (+0.9)
32kB	2560	0.0316	0.0631	0.1719	1.208 (+1.2)
16kB	5120	0.0273	0.0479	0.1172	3.857 (+1.5)
8kB	10240	0.0245	0.0382	0.0634	15.51 (+2.9)
4kB	20480	0.0201	0.0329	0.0657	66.09 (+8.5)
2kB	40960	0.0211	0.0290	0.0440	358.2 (+58.7)
256B [*]	327680	0.0218	0.0247	0.0271	>6h (+...)

Tabelle 3.1: Versuchsergebnisse: Laufzeit und Reception Overhead unter Verwendung verschiedener Blockgrößen mit einer 80MB-Quelldatei, jeweils 1000 Versuche pro Blockgröße, Laufzeit beinhaltet lediglich Lösungsfindung des Dekodieres, repräsentative Dauer einer Rekonstruktion der Quelldatei durch XOR($\oplus$)-Operation in Klammern angegeben, [*]: nur 10 Versuche aufgrund langer Rechenzeit

desto mehr SBs besteht die Quelldatei. Das hat zur Folge, dass der Berechnungsaufwand höher wird und sich somit die Laufzeit der Lösungsfindung verlängert. Im Gegenzug hierfür wird jedoch der durchschnittliche Reception Overhead verringert.

Ein Beispiel hierfür ist der direkte Vergleich der Ergebnisse für zwei verschiedene Blockgrößen. Bei der Wahl von 512kB großen Blöcke wird die Quelldatei in lediglich 160 SBs eingeteilt. Das hat zur Folge, dass ein mittlerer Reception Overhead von ∼77% für das erfolgreiche Dekodieren benötigt wird, die Lösungsfindung jedoch bereits in dem Bruchteil einer Sekunde abgeschlossen ist. Hingegen wird die Datei bei Wahl von 2kB großen Blöcken in 40960 SBs eingeteilt. Hieraus ergibt sich ein mittlerer Reception Overhead von nur ∼3%, jedoch dauert die Lösungsfindung ∼6min.

Der Reception Overhead bezieht sich hierbei lediglich auf die zu übertragenden Rohdaten. Für einen funktionierenden Dekodiervorgang werden jedoch nicht nur CB-Rohdaten benötigt, sondern auch deren Metadaten, also die verwendeten Seeds für die CBs. Um die nächsten Aussagen zu konkretisieren, wird folgende Annahme getroffen:

> Annahme: Wahl einer 160-bit CB-ID, nach [MM03a]

Bei der Wahl einer Blockgröße von 2kB oder mehr ist die Übertragung der CB-ID nahezu vernachlässigbar, da diese maximal ∼1% der Datenmenge ausmachen. Bei der Wahl einer Blockgröße von beispielsweise 256B ist der Anteil an Metadaten jedoch nicht mehr vernachlässigbar, da diese in dem Fall bereits ∼8% ausmachen. Somit sind die ermittelten Werte für 256B in der Tabelle eher theoretischer Natur, bedingt durch die extrem lange Berechnungszeit und den in der Realität viel höheren Reception Overhead, als die angegeben durchschnittlichen 2,47%. Genauer wäre der Overhead im gewählten Beispiel

$$0,0247 + 0,0755 = 0,1002$$

also ∼10%, anstatt der angegebenen 2,47%.

Der Autor von [MM03a] hat diese Metadaten aus seinen Betrachtungen bezüglich Reception Overhead ausgenommen, was die dort ermittelten Werte nicht praxisnah erscheinen lässt. Die Vermutung liegt nahe, dass das unbewusst geschehen ist, da kein Hinweis auf zusätzlichen Overhead durch Metadaten gegeben wird.

Im praktischen Einsatz von Online Codes sollte man jedoch nicht auf das eben beschriebene Problem treffen. Die sorgfältige Wahl einer geeigneten Blockgröße unter

Berücksichtigung aller beeinflussenden Faktoren ermöglicht den Einsatz des Kodierverfahrens ohne unerwartete Nebeneffekte. Tabelle 3.2 zeigt praxisnahe Laufzeiten des gesamten Vorgangs, d.h. vollständige Kodierung und Dekodierung, sowie Bildung der Prüfsummen von Quelldatei und rekonstruierter Datei.

Dateigröße	Blockgröße	SB-Anzahl	Laufzeit
800MB	64kB	12800	58s
800MB	32kB	25600	171s
80MB	4kB	20480	73s
80MB	2kB	40960	420s

Tabelle 3.2: Praktische Laufzeiten von kompletten Kodiervorgängen

3.3.3 Wahl der Blockgröße

Bei der Wahl der Blockgröße geht es darum, einen guten Kompromiss aus Reception Overhead und Berechnungsaufwand einzugehen. Die Blockgröße ist immer relativ zu der Dateigröße zu sehen. Somit ergibt eine Blockgröße von 4kB bei einer 400MB-Datei die selben Resultate, wie die Unterteilung einer 800MB-Datei in 8kB große Blöcke. Aus dem Grund wird im folgenden nur noch von der Blockanzahl gesprochen, in welche eine Datei zu unterteilen ist. Mithilfe der Daten aus dem vorigen Abschnitt ergeben sich folgende Richtlinien.

Bei ausreichender Größe der Quelldatei ist eine Unterteilung dieser in ca. 5.000-50.000 Blöcke sinnvoll. Hierbei muss darauf geachtet werden, ob Sender und Empfänger ausreichend gute Hardware besitzen um die benötigten Berechnungen schnell genug durchführen zu können. Wenn das der Fall ist, kann man sich in die Gegend der 50.000 Blöcke begeben, sollte das nicht der Fall sein, dann ist eine kleinere Blockanzahl von ca. 5.0000 sinnvoll. Falls es sich um eine kleine Datei handelt, also grob gesagt nicht größer als 1MB, ist eine Unterteilung dieser nur soweit sinnvoll, dass die CBs-IDs maximal ca. 10% der eigentlichen Blocklänge einnehmen. Je nach Anwendungsfall kann hier auch mit einem kleineren CB-ID-Raum gearbeitet

werden (z.B. 64bit statt 160bit). Kurz zusammengefasst kann man für einen einfachen Punkt-zu-Punkt Datentransfer über das Internet nach folgender Faustregel vorgehen:

Quelldatei *kB - 1MB: unterteilen in 100-500 Blöcke, kleinerer CB-ID-Raum

Quelldatei 1MB - 5MB: unterteilen in 500-1.000 Blöcke, ggf. kleinerer CB-ID-Raum

Quelldatei 5MB - 100MB: unterteilen in 1.000 - 10.000 Blöcke

Quelldatei 100MB - *GB: unterteilen in minimal 10.000 Blöcke, umso mehr Rechenleistung vorhanden, desto feinere Unterteilung, mehr als 50.000 Blöcke vermeiden

Diese Faustregel gilt für den Einsatz des Prototypens und repräsentiert lediglich Erfahrungswerte, sie basiert **nicht** auf wissenschaftlichem Hintergrund. Bei Implementierungen mit anderen Berechnungsgeschwindigkeiten oder anderen Transfer-Szenarien können die angegebenen Werte nicht sehr praxistauglich sein. Weiterhin muss z.B. in Broadcast-Szenarien geprüft werden, wie groß der CB-ID-Raum sein sollte. Wird dieser zu klein gewählt, kann es sein, dass Empfänger oft nutzlose Daten erhalten, was den Reception Overhead extrem ansteigen lässt.

3.3.4 Praxiseinsatz von Online Codes

Nachfolgend werden einige Szenarien für den gewinnbringenden Einsatz von Online Codes beschrieben. Jedoch gibt es auch Szenarien, wo der Einsatz mehr Nachteile als Vorteile mit sich bringt. Hierfür wird ein Negativbeispiel gebracht, wo die Verwendung von Online Codes in ihrer reinen Form nicht zielführend ist. Es werden jedoch mögliche Lösungen angerissen, welche noch separat untersucht werden müssen.

Vorweg kann jedoch gesagt werden, dass es mehrere Anwendungsfälle gibt, in denen sich der Einsatz von Online Codes auszahlt. Während der Entstehung dieser Arbeit war keine praktische Verwendung von Online Codes aufzufinden. Der Prototyp zeigt jedoch, dass das Verfahren auf beschriebenem Weg funktioniert und auch ausreichend schnell implementiert werden kann. Somit steht dem praktischen Einsatz, bezogen auf Funktionsfähigkeit und Leistungsfähigkeit, nichts im Weg.

3.3.4.1 Download mit unzuverlässiger Datenverbindung

Für Downloads von großen Datenmengen über eine unzuverlässige Datenverbindung eignen sich Online Codes hervorragend. Unter einer unzuverlässigen Datenverbindung wird hier eine Verbindung mit hoher Paketverlustwahrscheinlichkeit

verstanden. Hierfür ist ein klassisches Automatic Repeat reQuest (ARQ)-Verfahren nicht besonders gut geeignet, da ein sehr großer Kommunikationsoverhead zwischen Sender und Empfänger einer Datei entsteht. Ein Beispiel hierfür wäre der Transfer einer Datei über das Transmission Control Protocol (TCP). Diese Übertragungsweise hat sich im Internet zu einem oft verwendeten Standard etabliert.

Wesentlich effizienter ist hier eine User Datagram Protocol (UDP)-Übertragung der via Online Code kodierten Daten. Die genaue Wahrscheinlichkeit eines Paketverlustes rückt in den Hintergrund, solange sichergestellt ist, dass in absehbarer Zeit ausreichend CBs übertragen werden. So wird nur wenig Kommunikation zur Steuerung des Transfers benötigt und auch Verbindungsabbrüche stellen kein großes Problem dar. Der Empfänger muss lediglich eine Datei (ggf. erneut) anfordern und nach erfolgreicher Dekodierung dem Sender mitteilen, dass er keine weiteren Daten benötigt. Diese Methode ist beispielsweise prädestiniert für den Einsatz über Satellitenverbindungen, da der Rückkanal hier i.d.R. nicht sehr leistungsfähig ist und oft hohe Kosten verursacht.

3.3.4.2 Broad/-multicast großer Datenmengen

Durch den heutigen Fortschritt der Technik werden immer größere Datenmengen übertragen. In vielen Fällen handelt es sich hierbei jedoch um die selben Daten für eine Vielzahl von Empfängern. Ein Beispiel hierfür wäre ein Software-Update von Smartphones oder Fernsehgeräten. Ein Server, welcher eine Datei verteilt, muss für jeden Empfänger eine extra Session verwalten. Er muss für jeden einzelnen Empfänger bereit sein, von ihm Daten zu empfangen, um so eventuell verloren gegangene Pakete erneut zu senden.

Im Gegenzug ist der Transfer mithilfe von Online Codes insbesondere für den Server einfacher, da dieser alle Empfänger gleich behandeln kann. Den Empfängern ist es weitestgehend egal, welche Daten sie erhalten, es müssen lediglich Daten sein, welche sie noch nicht besitzen. Durch die Wahl eines ausreichend großen CB-ID-Raumes kann man vereinfacht davon ausgehen, dass diese Bedingung gegeben ist. Ein Server kann somit eine große Menge an Verwaltungsaufwand einsparen und seine verfügbare Bandbreite für den Upload effizienter nutzen.

3.3.4.3 Multi-Source-Download

Der Download einer Datei von einen Empfänger aus mehreren Quellen lässt sich ohne aufwändige Protokolle oder Verwaltungseinheiten realisieren. Ein Client muss lediglich von mehreren Servern eine Datei anfragen und im Anschluss die gleich-

großen CBs aus den verschiedenen Quellen empfangen. Es muss hierbei lediglich gewährleistet werden, dass die verschiedenen Server keine gleiche Abfolge von CB mit den selben IDs versenden. Das kann beispielsweise dadurch gesichert werden, dass diese einen PRNG mit einem möglichst exklusiven Seed initialisieren. Dieser Seed könnte beispielsweise aus der IP-Adresse eines Servers generiert werden.

Dieses Verfahren kann auch ohne Weiteres mit dem Broad/-multicast von Daten kombiniert werden um das Übertragungsmedium nicht unnötig auszulasten. Kombiniert man die beiden Verfahren miteinander, so hat man eine sehr effiziente Methode um Daten an mehrere Empfänger schnell und zuverlässig zu verteilen. Ein Ausfall von Servern beeinträchtigen Downloads lediglich in Sachen Geschwindigkeit, jedoch unter der Voraussetzung, dass mindestens ein Server verfügbar bleibt.

3.3.4.4 Komplexbeispiel

Ein Beispiel für das Zusammenkommen vieler ungünstiger Faktoren für einen klassischen Datentransfer ist die Verteilung von Kartendaten für Navigationsgeräte in Personenkraftwagen. Diese Faktoren sind

- Paketverluste
- Verbindungsabbrüche
- nicht vorhandener/schlechter/teurer Rückkanal
- gleiche Daten für viele Empfänger
- mehrere Quellen für gleiche Daten
- benötigte Transferzeit $>$ verfügbare Onlinezeit

Durch den Einsatz der vorher beschriebenen Techniken können diese Probleme jedoch einfach übergangen werden. Bei klassichen Transfermethoden müssten hier komplexe Protokolle entworfen werden. Sowohl Sender als auch Empfänger müssten einen sehr hohen Koordinierungsaufwand betreiben um einen Datentransfer erfolgreich abzuwickeln.

3.3.4.5 Tauschbörse

BitTorrent stellt aktuell eines der meist verwendeten Filesharing-Protokolle dar (nach [SM09]). Die Verteilung einer Datei erfolgt mittels des Transfers von sogenannten „pieces“ (siehe [Coh08]), welche zufällig ausgewählt werden. Diese „pieces“ sind nichts anderes als Blöcke der Quelldatei, den im Kontext der Arbeit sogenannten SBs. Ein möglicherweise auftretendes Problem bei der Verteilung von puren

SBs wird in nachfolgendem Beispiel dargestellt.

Angenommen, dass eine Quelle eine Datei anbietet und sich drei verschiedene Clients nacheinander im Verteilernetz befinden und jeweils zufällige 40% der Datei empfangen. Wenn die einzige Quelle danach die Verteilung einstellt und sich die drei Clients untereinander austauschen, ist es relativ unwahrscheinlich, dass sie die Datei vollständig rekonstruieren können, obwohl bereits 120% der Quelldaten im Netz transferiert wurden (siehe Sammelbilderproblem unter Abschnitt 2.1).

Eine mögliche Lösung für diese Szenario wäre der Einsatz von Online Codes. Die Quelle müsste CBs generieren und für jeden Client eine andere Abfolge von CB-IDs verwenden. Das wäre z.B. durch Zuweisung einer ID für jeden Client durch eine zentrale Einheit möglich. Denkbar wäre auch die Nutzung einer möglichst exklusiven Eigenschaft von Clients, wie beispielsweise deren IP-Adressen. Die Quelle könnte somit einen PRNG mit diesem Wert für jeden einzelnen Client initialisieren und somit verschiedene CB-ID-Abfolgen generieren. Wenn das sichergestellt ist, ist es durch Wahl einer geeigneten Blockgröße sehr wahrscheinlich, dass die Datei aus CBs von 120% der Datenmenge erfolgreich rekonstruiert werden kann.

Um den Verwaltungsoverhead gering zu halten, bietet sich eine Modifikation der CB-Generierung an. Der Inhalt eines CBs sollte nicht nur von einer ID abhängig sein, sondern außerdem von einer fortlaufenden Nummer. Es entstehen somit mehrere Streams an CBs, welche durch einen Seed identifiziert werden. CBs innerhalb eines Streams benötigen somit lediglich einen Index, sodass das Übertragen einer großen CB-ID entfällt. Teilnehmer in einem Verteilernetz können somit schnell deren Datenbestand austauschen, indem sie paarweise die Stream-ID und den zuletzt erhaltenen CB-Index verteilen. Abb. 3.11 zeigt einen möglichen Datenbestand eines Clients.

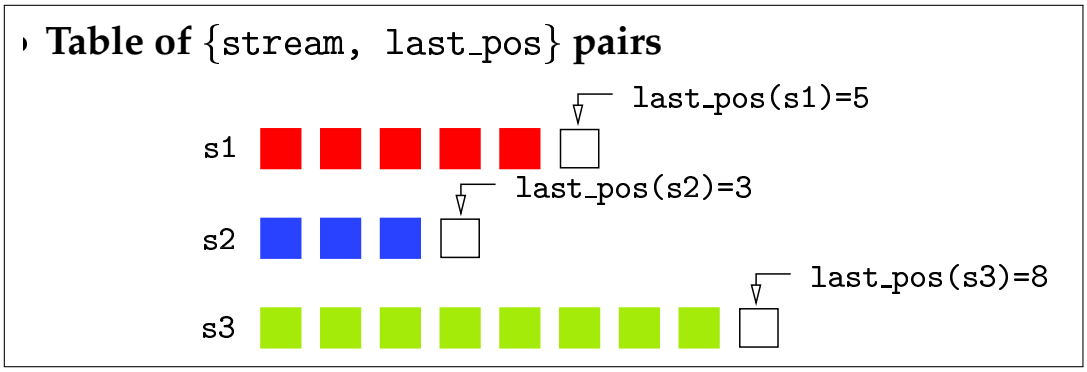

Abbildung 3.11: Beispiel: Datenbestand eines Tauschbörsen-Clients [MM03b]

3.3.4.6 Negativbeispiel: Streaming

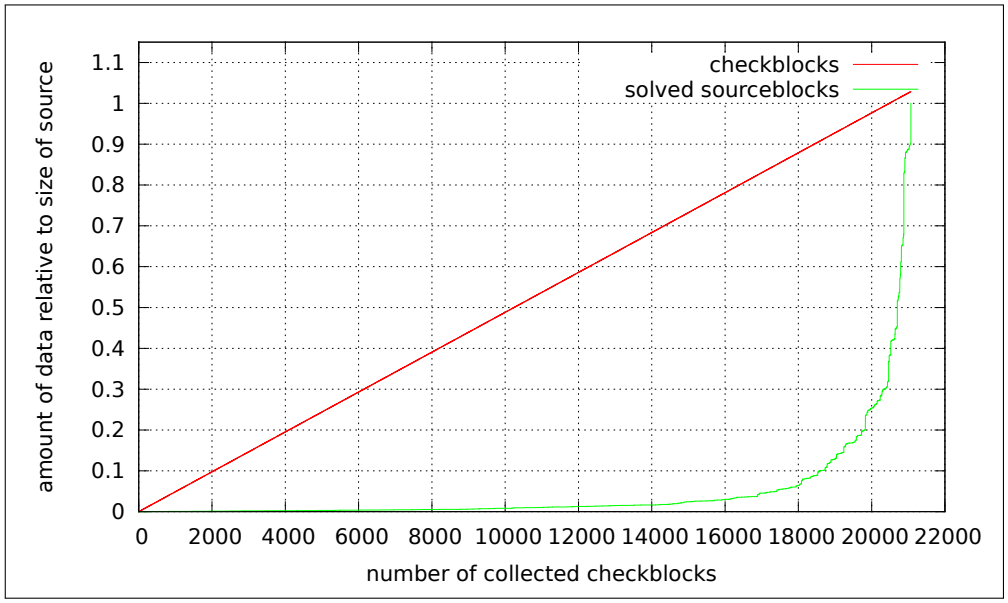

Abbildung 3.12: Beispiel Dekodierfortschritt, 80MB Quelldatei mit 4kB Blockgröße → 20480 SBs nach 21073 CBs rekonstruiert, ∼2,9% Overhead

In Abb. 3.12 wird der Dekodierfortschritt während des gesamten Dekodiervorgangs gezeigt. Die rote Linie stellt die Anzahl der empfangenen CBs dar, die grüne Linie die Anzahl der rekonstruierten SBs. Bis zu einer empfangenen Datenmenge von ∼90% der Quelldatei können maximal 10% der SBs rekonstruiert werden. Erst danach ist es dem Dekodierer möglich, einen Großteil der SBs zu rekonstruieren, bis nach dem Empfang des ∼1,03-fachen der Datenmenge die Datei vollständig dekodiert werden kann.

Das Verhalten ist damit zu begründen, dass es sich bei Online Codes um unsystematische Codes handelt, Der Empfänger einer Nachricht kann auch nicht voraussehen, welche Teile der Datei zu welchem Zeitpunkt rekonstruiert werden können. Diese Tatsache verbunden mit dem vorher beschriebenen Dekodierfortschritt führt dazu, dass sich Online Codes in ihrer reinen Form nicht für Streaming von beispielsweise Audio- oder Videodaten eignen.

Jedoch können verschiedene Lösungsansätze eventuell Abhilfe schaffen. Die folgenden Ansätze sind lediglich Überlegungen, welche noch nicht evaluiert wurden. Um deren Tauglichkeit zu prüfe, sind weitere Untersuchungen nötig, wofür der Prototyp eine gute Ausgangsbasis darstellt.

Blockweises Kodieren Unterteilung der Quelldatei in mehrere Blöcke, auf die jeweils getrennt ein Online Code angewendet wird. Die Wiedergabe eines Videos könnte somit ab dem Zeitpunkt erfolgen, wenn der erste Quelldatei-Block erfolgreich dekodiert wurde. Ein Problem hierbei könnte eine dauerhaft hohe Rechenlast darstellen um einen akzeptablen Reception Overhead zu erzielen.

Systematischer Online Code Angelehnt an Raptor Codes, welche systematisch sind, wäre es denkbar Online Codes zu einem systematischen Code umzuwandeln. Das würde bedeuten, dass die ersten 100% der übertragenen Daten exakt der Quelldatei entsprechen und erst im Anschluss daran CBs übertragen werden. Dieses Verfahren hätte den Vorteil, dass lediglich beim Auftreten von Paketverlusten Berechnungen zum Dekodieren nötig sind. Hierbei wäre zu klären, inwiefern sich die CBs mit deren Gradverteilung eignen um eine Datei mit möglichst wenig Overhead zu dekodieren. Für das Streaming von zeitkritischen Daten ist diese Methode jedoch nicht sinnvoll, da die Wiederherstellungsinformationen (CBs) erst nach den Quelldaten übertragen werden, weswegen sich nachfolgendes Hybridverfahren vermutlich besser eignet.

Hybridverfahren Kombination von blockweiser Kodierung und dem Übergang zu systematischen Codes. Hiermit könnte es durch die systematische Komponente möglich sein, die Rechenlast drastisch zu reduzieren und den Reception Overhead dennoch in akzeptablen Grenzen zu halten.

3.3.4.7 Datenaufbewahrung

Online Codes können nicht nur zur Datenübertragung verwendet werden, sondern beispielsweise auch zur verlässlichen Datenaufbewahrung. Das kann dadurch erreicht werden, dass die zu archivierenden Daten kodiert werden und eine Menge an CBs auf mehreren Datenträgern verteilt gespeichert werden. Auch eine verteilte Speicherung der Daten auf einem Datenträger kann in einigen Fällen hilfreich sein. Beispielsweise kann es bei Festplatten zum Defekt von einzelnen Sektoren kommen, sodass nur ein Teil von Daten einer Festplatte verloren geht.

Verbindet man die verteilte Speicherung auf einem Datenträger mit dem Streuen der Daten auf mehrere Datenträger, so erhält man ein sehr leistungsfähiges und effizientes Archivierungsverfahren. Hierfür müsste jedoch ein Schema erarbeitet werden, welche Menge von Daten an welchem Ort abgespeichert werden sollte, um ein möglichst ausfallsicheres und dennoch speicherplatzsparendes Verfahren umzusetzen. Für einen häufigen Zugriff auf die Daten eignet sich das Verfahren jedoch weniger, da ein vollständiger Dekodiervorgang durchgeführt werden muss, bevor die Daten zur Verfügung stehen.

4 Zusammenfassung und Ausblick

Durch die Betrachtung der ratenlosen Codes wurden Ähnlichkeiten und Unterschiede erarbeitet, sodass ein direkter Vergleich der Verfahren vorgenommen werden konnte. Dieser Vergleich zeigt eindeutige Vor- und Nachteile der einzelnen Verfahren. Auf Basis dieser Übersicht kann relativ schnell festgestellt werden, welche der Verfahren sich für ein mögliches Vorhaben eignen.

Durch die Patentbelastung der anderen Verfahren wurde sich dafür entschieden, einen Prototypen auf Grundlage von Online Codes umzusetzen. Dieser Prototyp stellt ein gutes Fundament für den praktischen Einsatz von Online Codes dar. Außerdem ermöglicht er weitere wissenschaftliche Untersuchungen in dem Themengebiet. Es kann mit ihm beispielsweise die Eignung von Online Codes in weiteren praktischen Szenarien geprüft werden.

Mithilfe des Prototypens war es bereits in dieser Arbeit möglich, einige praktische Szenarien zu untersuchen und somit eine Einschätzung der Leistungsfähigkeit von Online Codes vorzunehmen. Die Implementierung hat gezeigt, dass die Laufzeiten des Prototypens für viele Anwendungen in akzeptablen Bereichen liegen. Hierfür wird jedoch eine angemessene Wahl der Parameter vorausgesetzt. Dafür wurde eine Faustregel angegeben, welche die Wahl, durch Eingrenzung möglicher Parameter, erleichtern soll.

Für die Parameter kann jedoch nicht pauschal ein Wert angegeben werden, da diese von mehreren Faktoren beeinflusst werden, wie z.B.

- verfügbare Rechenkapazitäten des Senders
- verfügbare Rechenkapazitäten des Empfängers
- akzeptierte Rechenzeit
- maximaler Übertragungsoverhead
- zugrunde liegende Datenmenge

Weiterhin wurden, neben dem Datentransfer mit einer schlechten Verbindung, mögliche Anwendungen von Online Codes aufgezeigt. Wenn mehrere Clients eine große Datenmenge von einem oder mehreren Servern anfragen, kann gegenüber klassischen Verteilungsverfahren eine bessere Leistung bei weniger Verwaltungsaufwand erzielt werden. Es wurde auch gezeigt, welche Vorteile Online Codes beim Einsatz in Tauschbörsen bringen und wie deren Integration umsetzbar wäre.

Jedoch wurden auch Szenarien aufgeführt, für welche sich andere Techniken besser eignen bzw. eine Modifikation des Verfahrens nötig ist. Es wurde ein Lösungsansatz für das Streaming von zeitkritischen Daten entworfen, welcher jedoch noch geprüft werden muss. Weiterhin wurden keine Betrachtungen zum praktischen Einsatz auf Geräten mit begrenzten Ressourcen vorgenommen, sondern lediglich der Einsatz auf leistungsfähiger Computer-Hardware. Durch den großen technischen Fortschritt von mobilen Geräte, wie beispielsweise Smartphones, ist jedoch auch ein Einsatz von Online Codes in diesem Bereich denkbar.

Abschließend kann gesagt werden, dass, bedingt durch den technischen Fortschritt heutiger Hardware, Online Codes effizient in der Praxis angewendet werden können und in vielen Fällen enorme Vorteile mit sich bringen.

Anhang

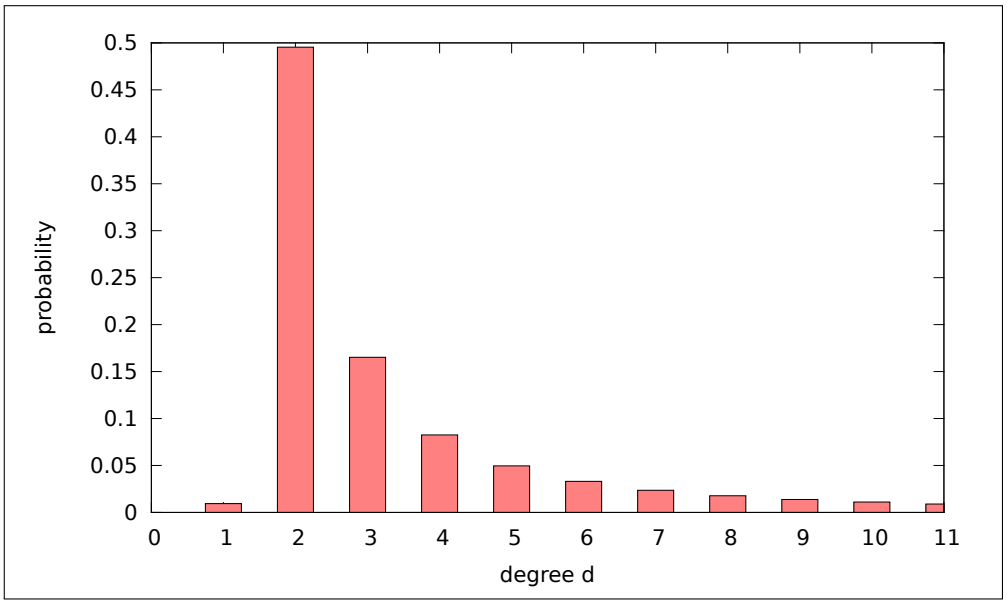

Abbildung A.1: Beispiel: Ausschnitt Wahrscheinlichkeitsverteilung für niedrige CB-Grade mit $\epsilon = 0.01$ und $q = 3$

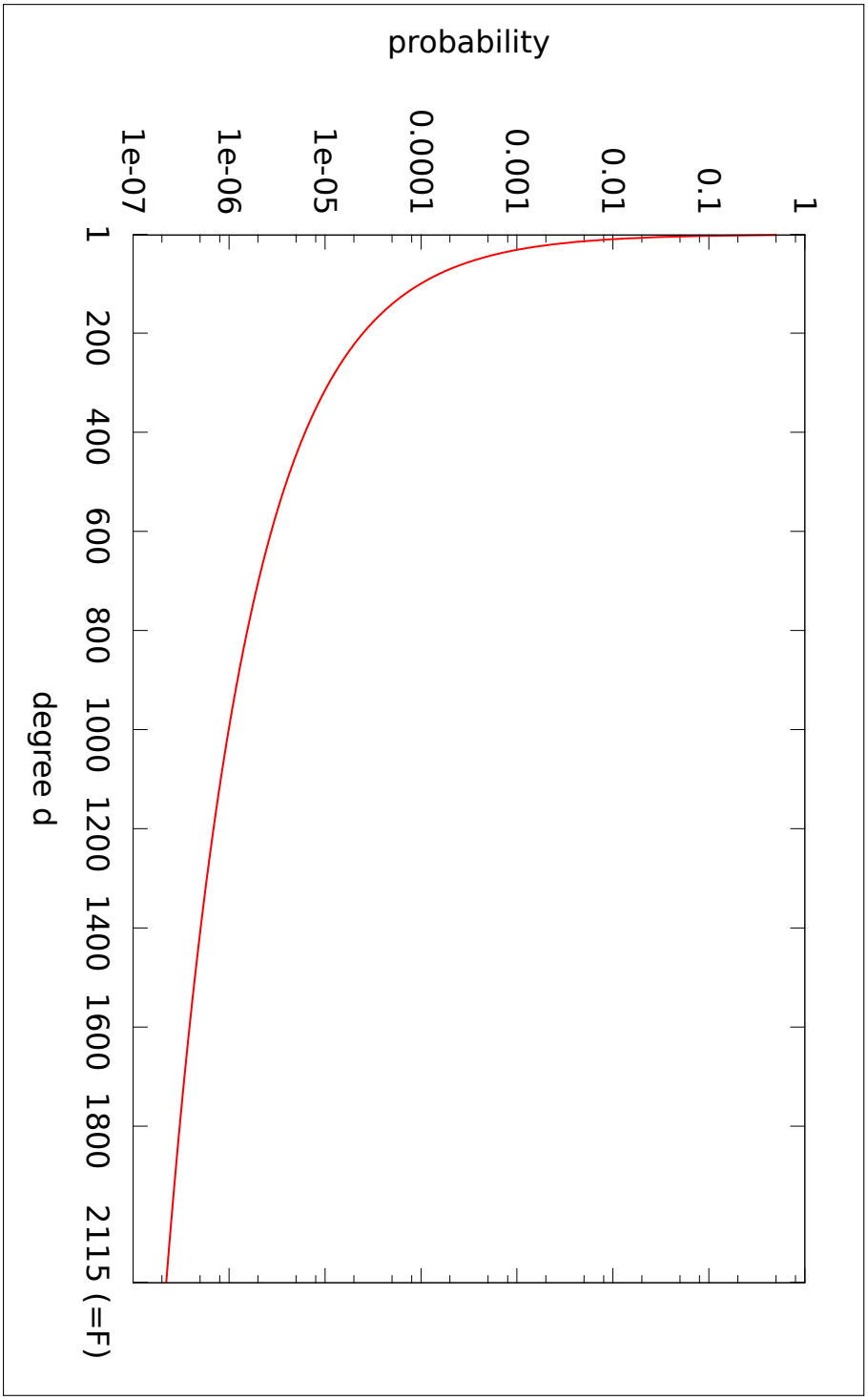

Abbildung A.2: Beispiel: Wahrscheinlichkeitsverteilung für CB-Grade mit $\epsilon = 0.01$ und $q = 3$

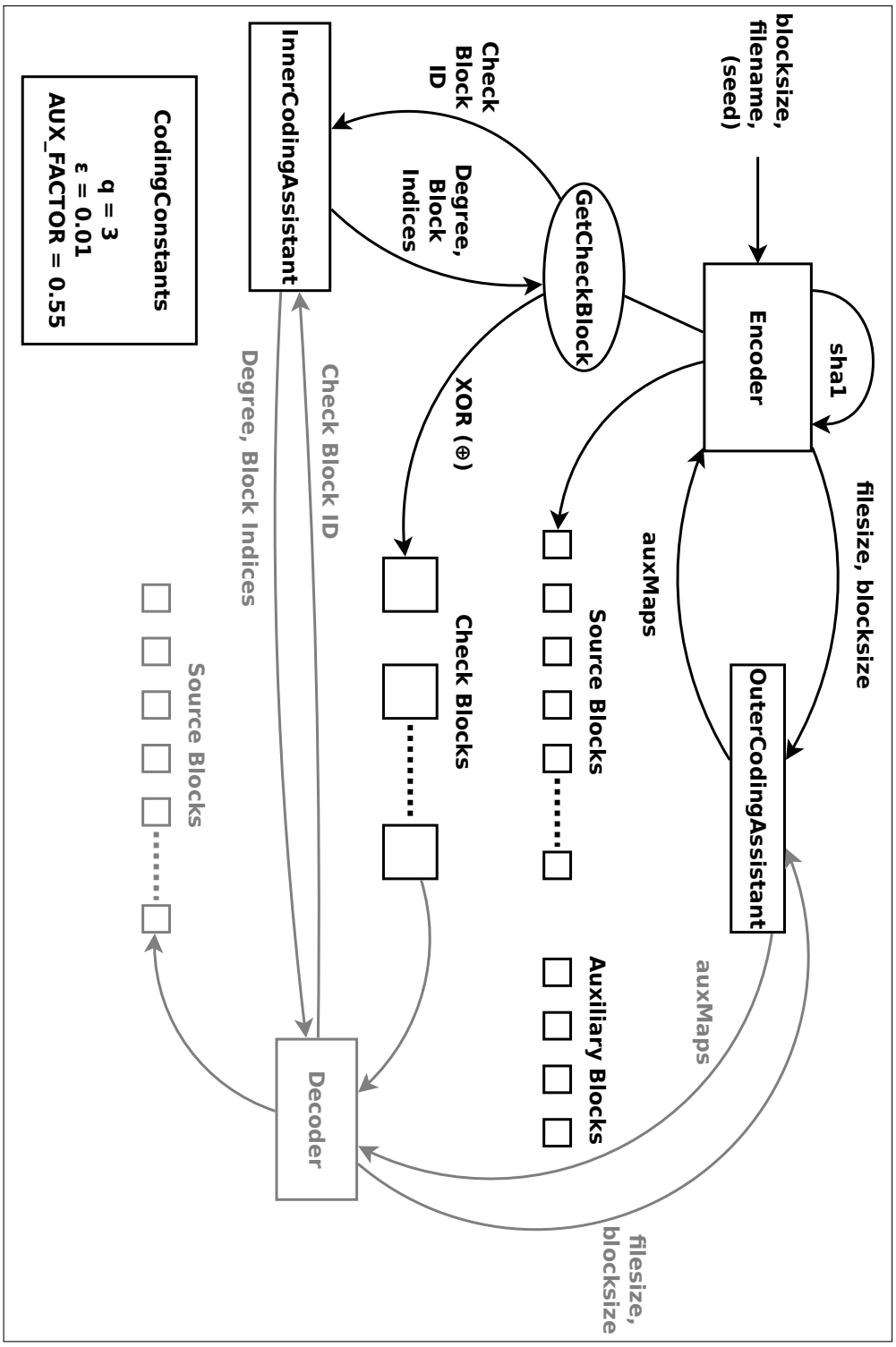

Abbildung A.3: Architektur des Prototypens – Encoding

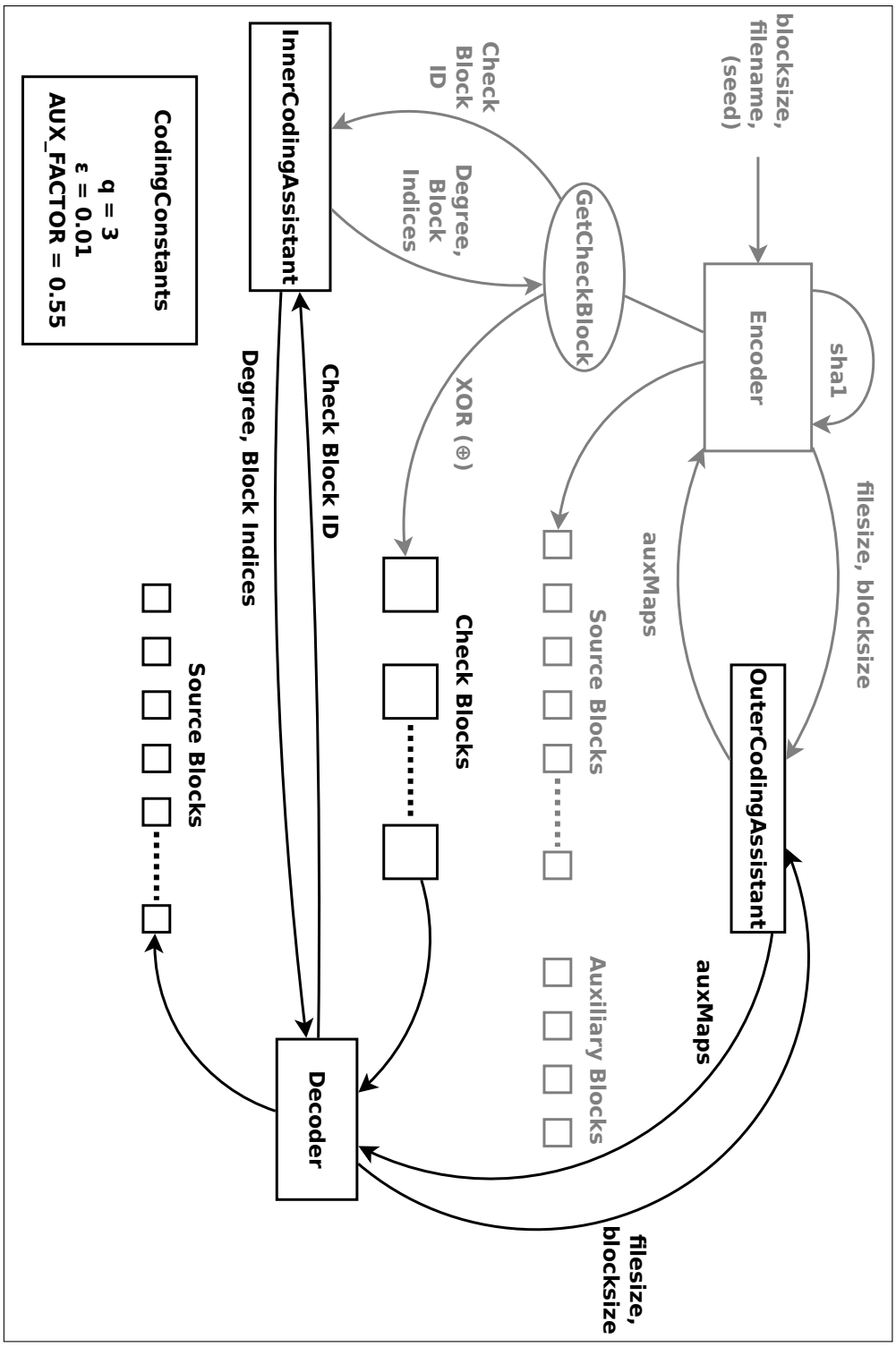

Abbildung A.4: Architektur des Prototypens – Decoding

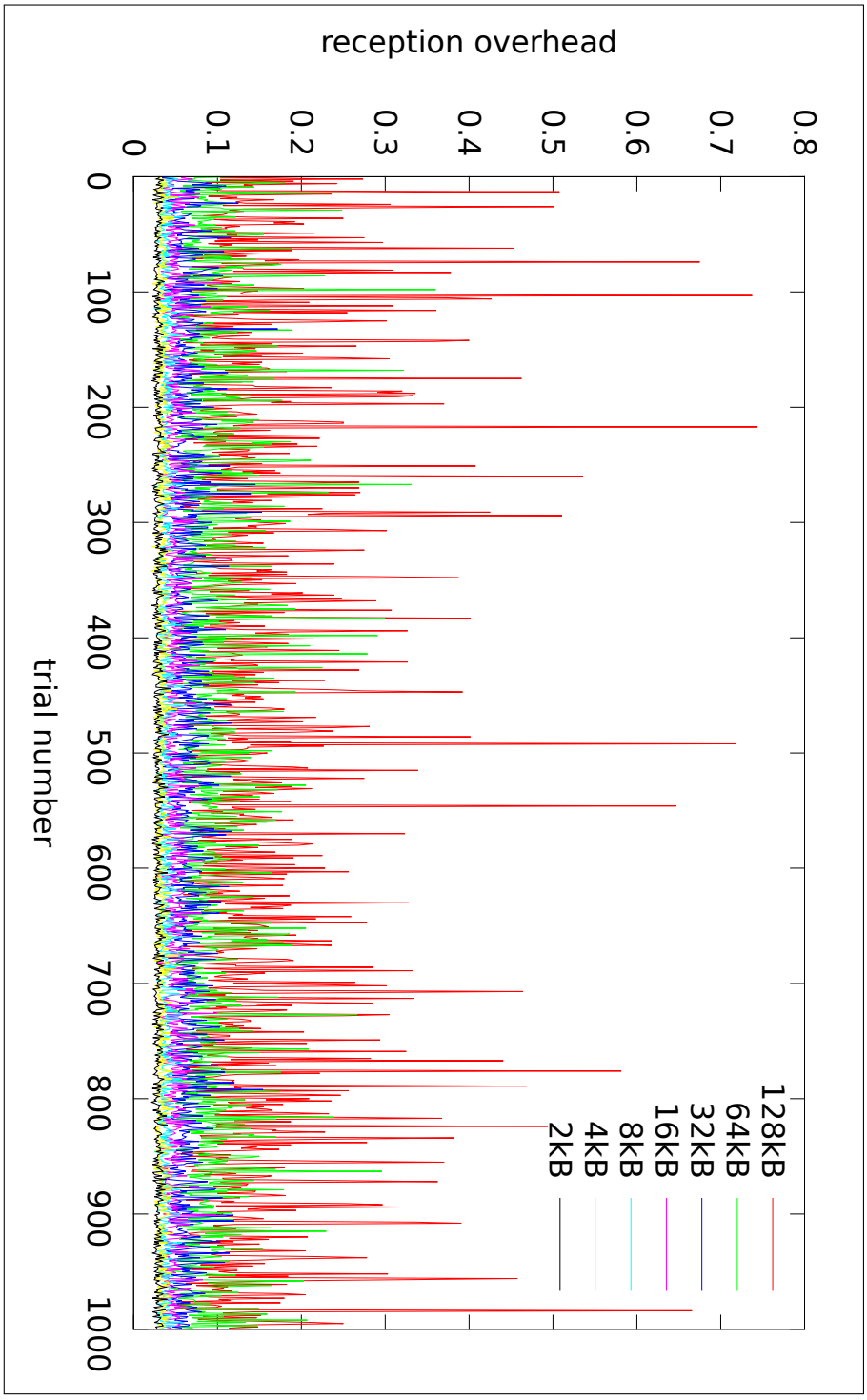

Abbildung A.5: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit verschiedenen Blockgrößen (2kB, 4kB, 8kB, 16kB, 32kB, 64kB und 128kB)

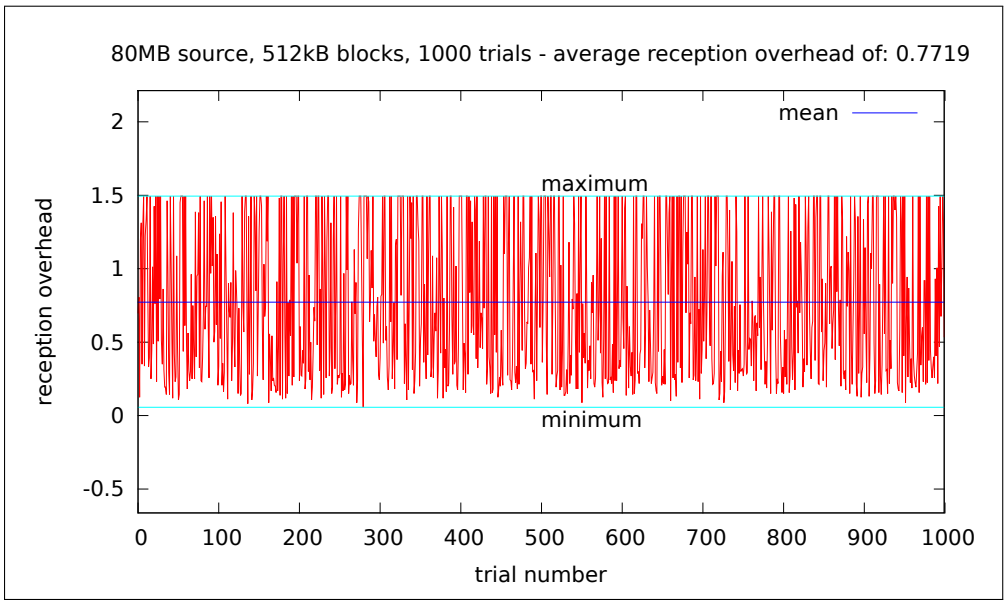

Abbildung A.6: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 512kB Blöcken

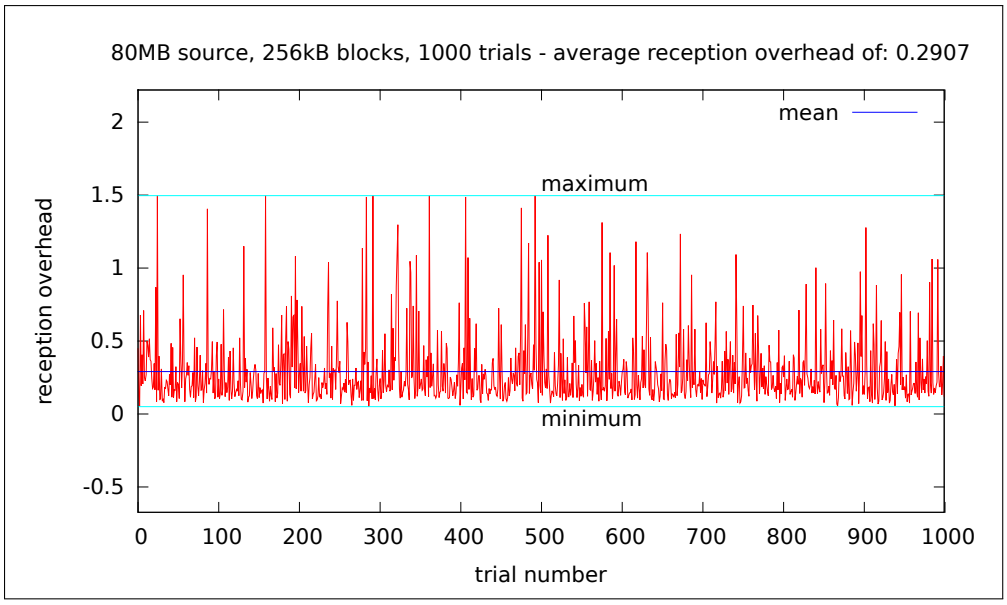

Abbildung A.7: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 256kB Blöcken

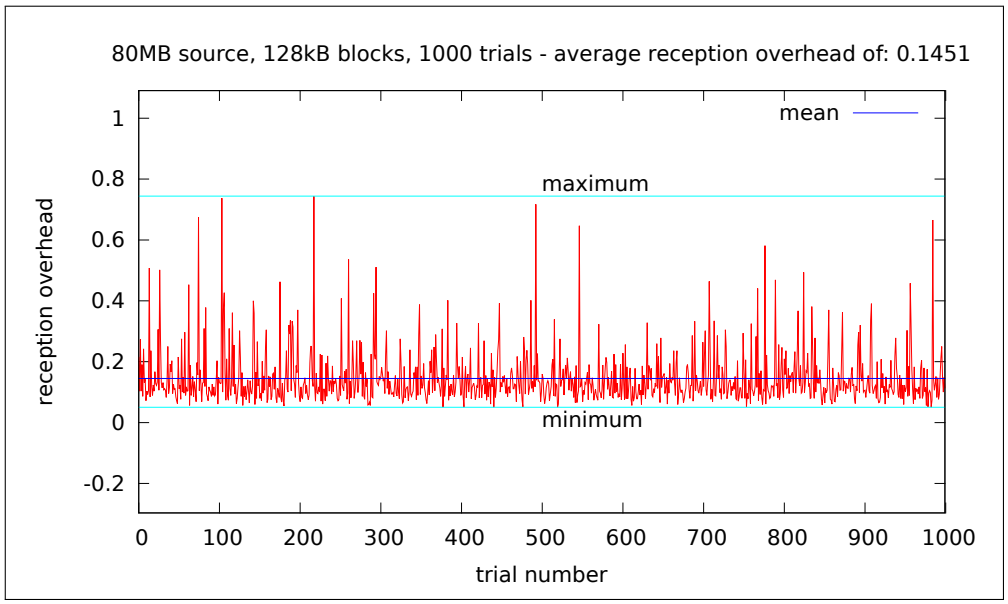

Abbildung A.8: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 128kB Blöcken

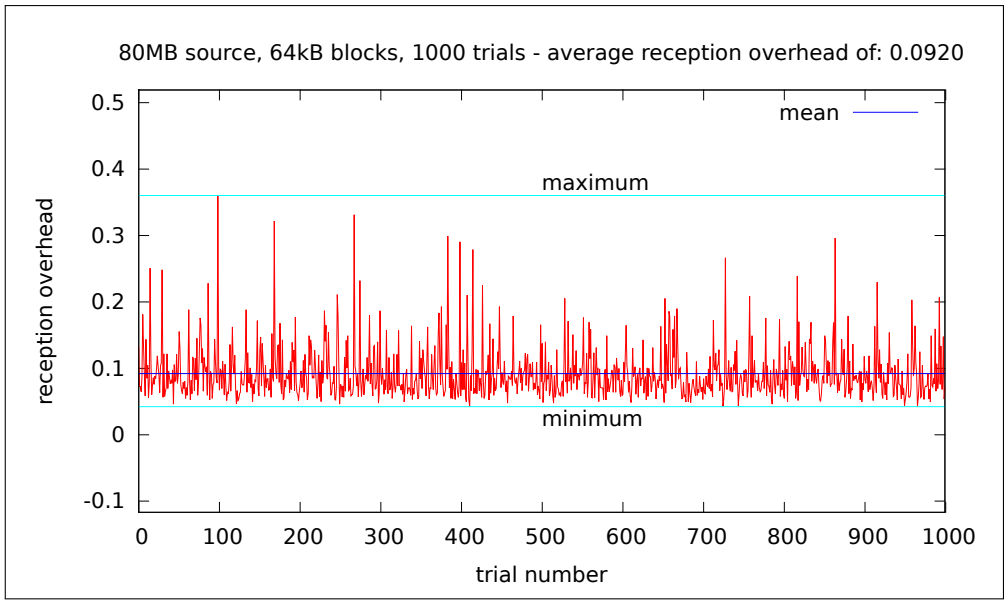

Abbildung A.9: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 64kB Blöcken

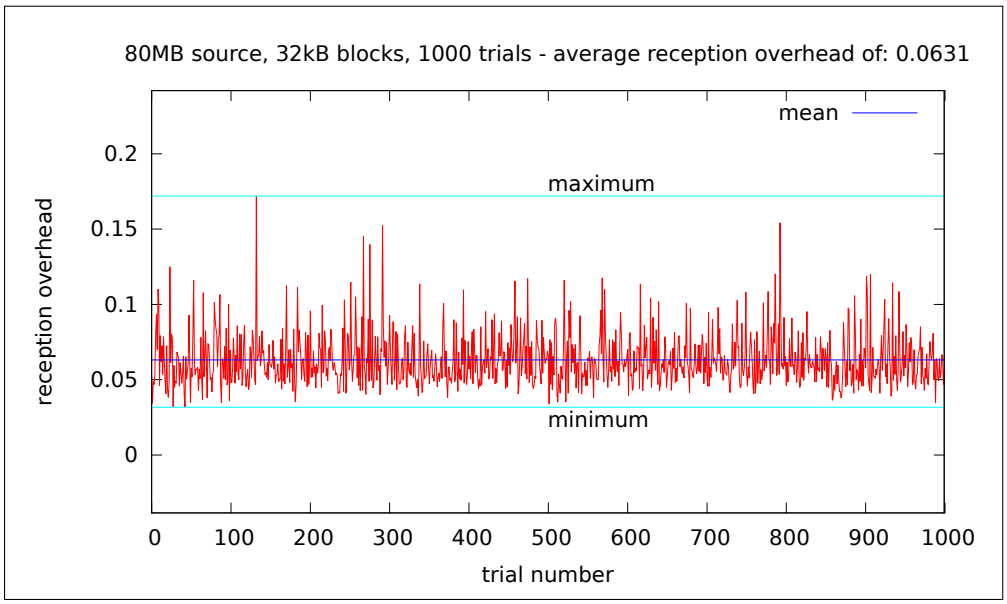

Abbildung A.10: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 32kB Blöcken

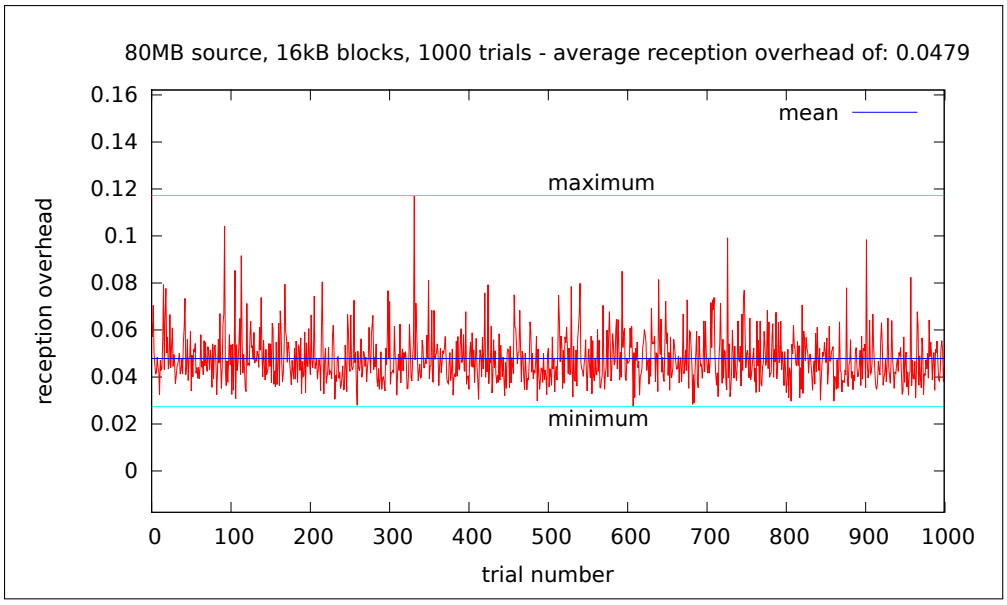

Abbildung A.11: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 16kB Blöcken

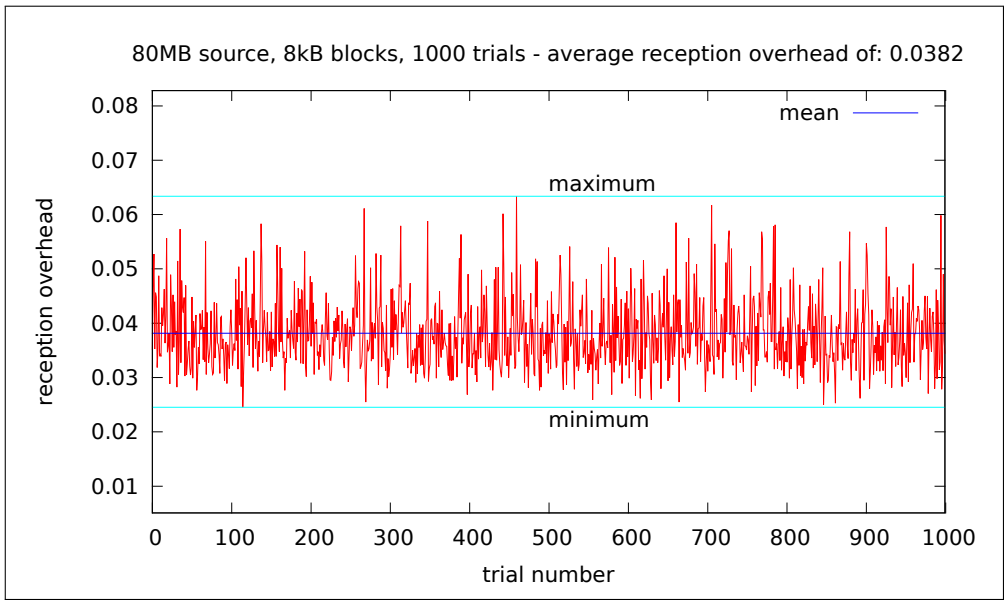

Abbildung A.12: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 8kB Blöcken

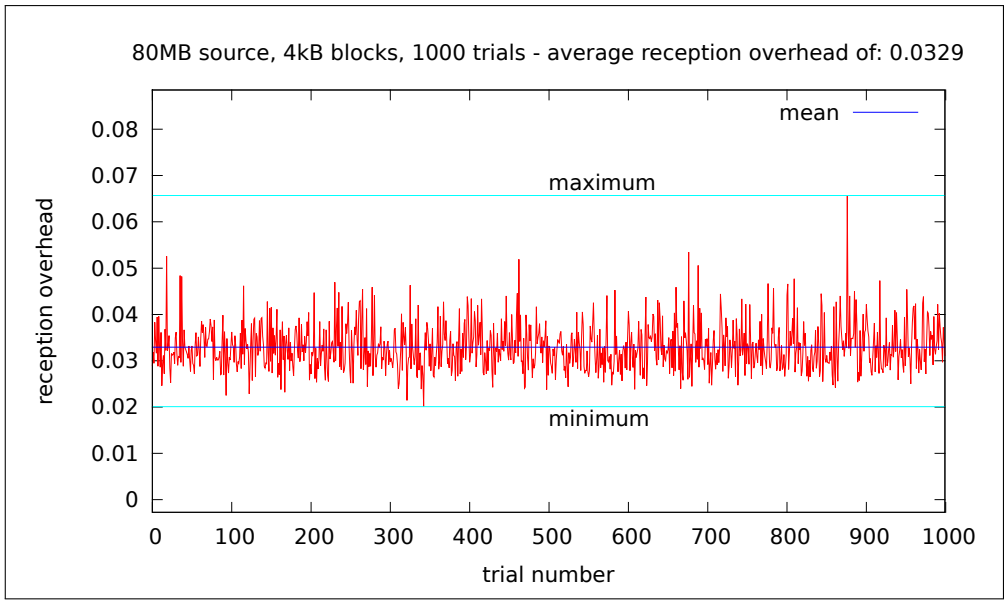

Abbildung A.13: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 4kB Blöcken

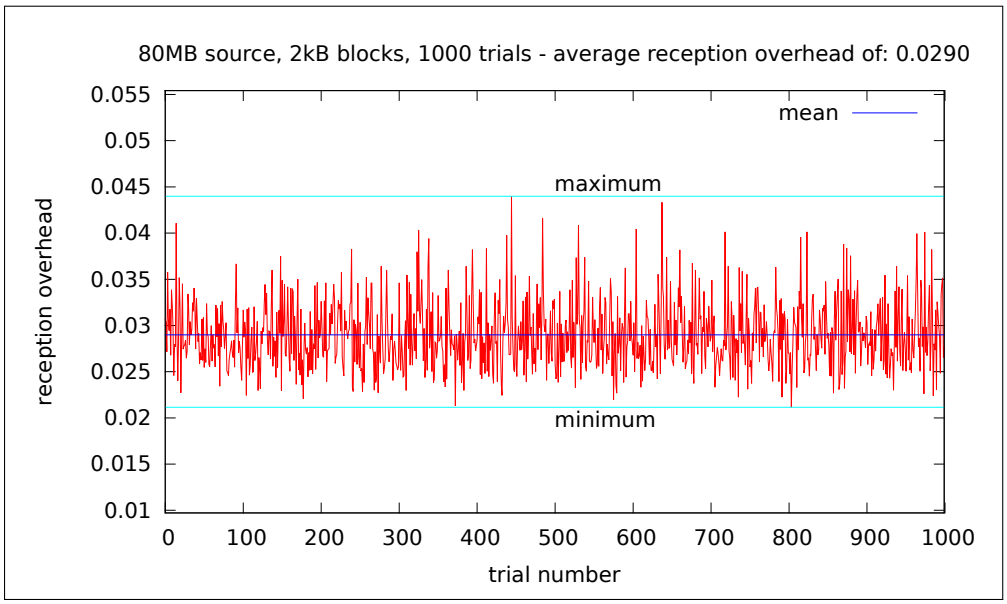

Abbildung A.14: Versuchsergebnisse: Reception Overhead über jeweils 1.000 Versuche beim Dekodieren einer 80MB-Datei mit 2kB Blöcken

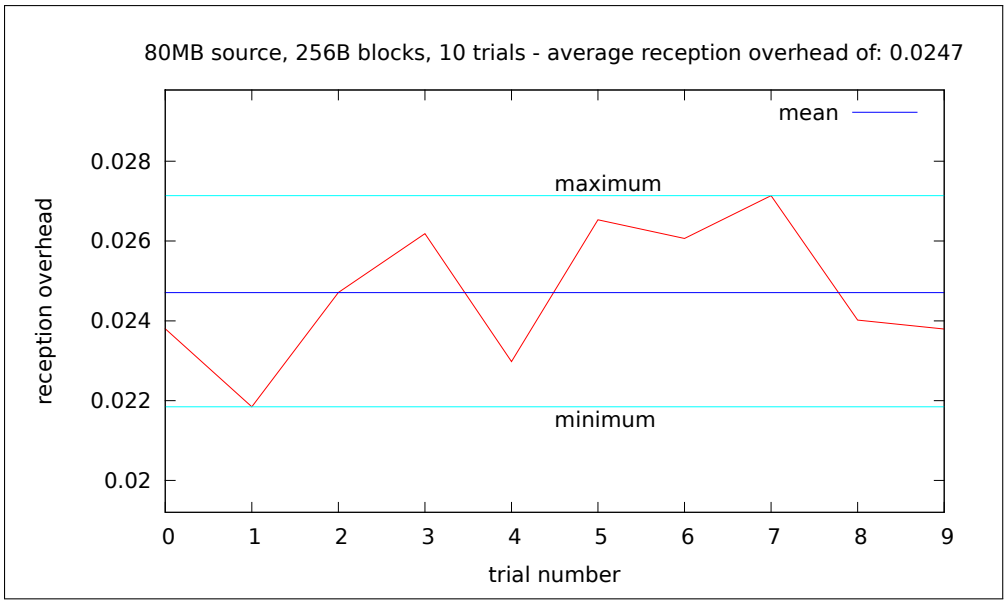

Abbildung A.15: Versuchsergebnisse: Reception Overhead über jeweils 10 Versuche beim Dekodieren einer 80MB-Datei mit 256B Blöcken

Quellenverzeichnis

[Bar13] Bard, Adam: *Top Github Languages for 2013 (so far).* URL `http://adambard.com/blog/top-github-languages-for-2013-so-far/`; (abgerufen am 20.11.2013), August 2013

[Coh08] Cohen, Bram: *The BitTorrent Protocol Specification.* `http://www.bittorrent.org/beps/bep_0003.html`; (abgerufen am 01.12.2013), 2008

[Cor13] Corporation, Valve: *Steam: Game and Player Statistics.* `http://store.steampowered.com/stats/content/`; (abgerufen am 02.12.2013), December 2013

[Gal63] Gallager, Robert G.: *Low-Density Parity-Check Codes.* 1963

[LMS+97] Luby, Michael G. ; Mitzenmacher, Michael ; Shokrollahi, M. A. ; Spielman, Daniel A. ; Stemann, Volker: Practical loss-resilient codes. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.* New York, NY, USA : ACM, 1997 (STOC '97). – ISBN 0–89791–888–6, 150–159

[LPSK13] Lohmar, Thorsten ; Puustinen, Stig ; Slssingar, Michael ; Kenehan, Vera: Delivering content with LTE Broadcast. In: *Ericsson Review* 1 (2013), February, S. 8

[LSW+11] Luby, M. ; Shokrollahi, A. ; Watson, M. ; Stockhammer, T. ; Minder, L.: *RaptorQ Forward Error Correction Scheme for Object Delivery.* RFC 6330 (Proposed Standard). `http://www.ietf.org/rfc/rfc6330.txt`. Version: August 2011 (Request for Comments)

[LSWS07] Luby, M. ; Shokrollahi, A. ; Watson, M. ; Stockhammer, T.: *Raptor Forward Error Correction Scheme for Object Delivery.* RFC 5053 (Proposed Standard). `http://www.ietf.org/rfc/rfc5053.txt`. Version: Oktober 2007 (Request for Comments)

[Lub01] Luby, Berkeley C. Michael G. G. Michael G.: *Information additive code generator and decoder for communication systems*. Patent. http://www.patentlens.net/patentlens/patent/US_6307487/en/. Version: 10 2001. – US 6307487

[Lub02] Luby, M.: LT codes. In: *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, 2002. – ISSN 0272–5428, S. 271–280

[Mata] Matsumoto, Makoto: *Mersenne Twister: A random number generator (since 1997/10)*. URL http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html; (abgerufen am 25.11.2013),

[Matb] Matsumoto, Makoto: *What & how is MT?* URL http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ewhat-is-mt.html; (abgerufen am 25.11.2013),

[May02] Maymounkov, Petar: Online codes (Extended Abstract) / New York University. 2002. – Forschungsbericht

[MM03a] Maymounkov, Petar ; Mazières, David: Rateless Codes and Big Downloads. In: Kaashoek, M. F. (Hrsg.) ; Stoica, Ion (Hrsg.): *IPTPS* Bd. 2735, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–40724–3, 247-255

[MM03b] Maymounkov, Petar ; Mazières, David: *Rateless Codes and Big Downloads - Talk slides*. http://pdos.csail.mit.edu/~petar/pubs.html. Version: 2003

[PTC13] Pandya, Avani U. ; Trapasiya, Sameer D. ; Chinnam, Santhi S.: Comparative Analysis of AL-FEC Raptor and RaptorQ over 3GPP eMBMS Network. In: *International Journal of Electronics and Communication Engineering* 2 (2013), May, Nr. 2, S. 169–178

[RU08] Richardson, Tom ; Urbanke, Ruediger: *Modern Coding Theory*. New York, NY, USA : Cambridge University Press, 2008. – ISBN 0521852293, 9780521852296

[Sha01] Shannon, C. E.: A mathematical theory of communication. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 5 (2001), Januar, Nr. 1, 3–55. http://dx.doi.org/10.1145/584091.584093. – DOI 10.1145/584091.584093. – ISSN 1559–1662

[Sho06] Shokrollahi, Amin: Raptor codes. In: *IEEE/ACM Trans. Netw.* 14 (2006), Juni, Nr. SI, 2551–2567. http://dx.doi.org/10.1109/TIT.2006.874390. – DOI 10.1109/TIT.2006.874390. – ISSN 1063–6692

[SL05] Shokrollahi, M. A. ; Luby, Michael G.: *Systematic encoding and decoding of chain reaction codes*. Patent. http://www.patentlens.net/patentlens/patent/US_6909383/en/. Version: 06 2005. – US 6909383

[SL11] Shokrollahi, A. ; Luby, M.: *Raptor Codes*. Now Publishers Inc, 2011 (Foundations and trends in communications and information theory). http://books.google.de/books?id=JX7g-5PdFRIC. – ISBN 9781601984463

[SLK05] Shokrollahi, M. A. ; Lassen, Soren ; Karp, Richard: *Systems and processes for decoding chain reaction codes through inactivation*. Patent. http://www.patentlens.net/patentlens/patent/US_6856263/en/. Version: 02 2005. – US 6856263

[SM09] Schulze, Hendrik ; Mochalski, Klaus: *Internet Study 2008/2009*. February 2009

[Sno01] Snoeren, Alex C.: *On the Practicality of Low-Density Parity-Check Codes*. 2001

[Ven12] Venkiah, Auguste: *Analysis and Design of Raptor Codes for Multicast Wireless Channels*, University of Cergy-Pontoise, Diss., 2012

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

_____

Martin Dönicke