

Bachelorarbeit zum Thema

”Untersuchung und Implementierung von
Erasure Resilient Codes für eine
P2P-basierte Speicherplattform”

zur Erlangung des akademischen Grades

Bachelor Of Science

vorgelegt dem

Institut für Angewandte Mikroelektronik und Datentechnik
der Universität Rostock

Maik Gotzmann

15. Oktober 2009

Thema für eine Bachelorarbeit

für Herrn Maik Gotzmann

Untersuchung und Implementierung von Erasure Resilient Codes für eine P2P-basierte Speicherplattform

Die Erfassung und Kombination großer Datensätze und –dienste erlangen auf Grund steigender Anforderungen an Dienstgüte und Sicherheit immer höhere Bedeutung für Netzbetreiber im Telekommunikationsbereich. Insbesondere die Speicherung von Logdateien und statistischen Daten bezüglich Netzwerkparametern und Nutzerinformationen steigt stark an, da sowohl Datenverkehr als auch die Anzahl unabhängiger Datenflüsse sprunghaft zunehmen. Die Backbones der Netzbetreiber generieren an jedem Tag Hunderte Gigabyte an Logdateien und Statistiken, die gespeichert und analysiert werden müssen.

Um solche Datenmengen verarbeiten zu können, wird eine hochgradig dezentrale Speicherplattform benötigt. Eine verteilte Peer-to-Peer (P2P)-basierte Plattform stellt hierbei eine vielversprechende Lösung mit hoher Performance und Robustheit dar, die existierende zentrale Lösungen ersetzen kann. Dabei muss eine hohe Verfügbarkeit der zu speichernden Daten gewährleistet werden, um Ausfälle von P2P-Knoten kompensieren zu können. Hierzu bieten sich Erasure Resilient Codes an, die eine effiziente verschränkte Datenspeicherung erlauben.

Im Zuge dieser Arbeit sollen Erasure Resilient Codes untersucht werden und mindestens ein geeigneter Algorithmus in Software zu implementieren. Abschließend ist der implementierte Algorithmus zu testen und zu verifizieren.

Zusammenfassend sind folgende Aufgaben zu lösen:

- Einarbeitung in Prinzipien der verteilten Datenspeicherung und Erasure Resilient Codes
- Auswahl mindestens eines Algorithmus zur Realisierung von Erasure Resilient Codes
- Implementierung mindestens eines Algorithmus zur Realisierung von Erasure Resilient Codes in Software
- Verifikation des implementierten Algorithmus

Betreuer: Dipl.-Ing. Peter Danielis

Tag der Ausgabe: 17.08.2009

Tag der Abgabe: 17.10.2009

Prof. Dr. D. Timmermann
Betreuender Hochschullehrer

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Aufbau der Bachelorarbeit	2
2	Grundlagen und Konzept der verteilten Speicherung	3
2.1	Kriterien zur verteilten Speicherung in P2P-Netzwerken	3
2.2	Speicherung mittels Replikation	4
2.3	Speicherung durch Verwendung von Erasure Resilient Coding . . .	6
2.4	Replikation vs. Erasure Resilient Coding	8
3	Algorithmen zur Realisierung der Erasure Resilient Codes	10
3.1	Grundlagen der Codierungstheorie	10
3.2	Klassifizierung und Kurzbeschreibung von Erasure Resilient Codes	12
3.3	Reed-Solomon Codes	15
3.3.1	Grundlagen und Kenngrößen	15
3.3.2	Algorithmus zum Encoding	17
3.3.3	Algorithmus zum Decoding	21
4	Implementierung und Verifikation	24
4.1	Die Entwicklungsumgebung Qt	24
4.2	Das Hauptfenster: Bedienelemente und Funktionen	26
4.3	Unterprogramm: Select File	28
4.4	Unterprogramm: Split	29
4.5	Unterprogramm: Encode	30
4.5.1	Erstellung des Erweiterungskörpers	30
4.5.2	Berechnung des Generatorpolynoms	33
4.5.3	Codiervorgang	35
4.6	Unterprogramm: Loss	36
4.7	Unterprogramm: Decode	37
4.8	Verifikation	40
4.9	Datendurchsatz	41

4.10 Komplexität Encoding und Decoding	43
5 Zusammenfassung und Ausblick	44
6 Erklärung über die selbständige Anfertigung der Arbeit	47

Abbildungsverzeichnis

2.1	Grundprinzip der verteilten Speicherung mittels Replikation ($S = 2$)	4
2.2	Grundprinzip der verteilten Speicherung mittels Erasure Resilient Codes	6
2.3	Grundprinzip der Codierung mittels Erasure Resilient Codes . . .	7
3.1	Klassifizierung von Error Correction Codes	12
3.2	Schema des Systematischen Blockcodes	13
4.1	IDE von Qt V.4.5.2	25
4.2	Bedienelemente und Funktionen des Hauptfensters	28
4.3	Grundprinzip der Dateiteilung bei $mchunks = 3$	29
4.4	Berechnung des Restpolynoms am Beispiel von α^{10}	31
4.5	Grund- und Erweiterungskörper über $GF(2^8)$	32
4.6	Algorithmus zur Erzeugung des Generatorpolynoms	34
4.7	Schemata des Codierungsvorganges bei: $m = 3, k = 2, n = 5$. . .	35
4.8	Dialog zum Löschen einiger Dateiblöcke	36
4.9	Schemata der Decodierung bei: $m = 3, k = 2, n = 5$	38
4.10	Gauß-Algorithmus im Falle des Fehlens von drei Chunks	39
4.11	Beispiel einer binären Codierung mit RS(32,16)-Code	41
4.12	Codierungs-/Decodierungszeiten am Beispiel eines RS(16+x,16)-Codes	43

Tabellenverzeichnis

2.1	Grundlegende Kenngrößen der verteilten Speicherung	4
2.2	Datenverfügbarkeit im Falle der Replikation	6
2.3	Datenverfügbarkeit im Falle des Erasure Resilient Codings	8
3.1	Grundlegende Kenngrößen der Kodierungstheorie	12
3.2	Ausgewählte Modularpolynome	17
3.3	Erweiterungskörper	18
4.1	Datendurchsatz des Encoding-Algorithmus	42
4.2	Datendurchsatz des Decoding-Algorithmus	42

Abkürzungsverzeichnis

ASIC	:	Application Specific Integrated Circuit
ECC	:	Error Correction Code
FIFO	:	First In – First Out
FPGA	:	Field Programmable Gate Array
HDL	:	Hardware Description Language
IDE	:	Integrated Development Environment
LFSR	:	Linear Feedback Shift Register
LSS	:	Least Significant Symbol
MDS	:	Maximum Distance Separable
MSS	:	Most Significant Symbol
MTBF	:	Mean Time Between Failures
MTTR	:	Mean Time To Recover
P2P	:	Peer To Peer
RAID	:	Redundant Array of Inexpensive Disks
XOR	:	eXclusive OR

1 Einleitung

1.1 Motivation

Durch das exponentielle Wachstum an Rechenleistung und Speicherkapazität im Bereich der Rechentechnik, wächst auch die zu verarbeitende Datenmenge im privaten als auch im wirtschaftlichen Bereich. Netzwerke, die den Austausch dieser sicherstellen und der Kommunikation dienen, sind aus der Gegenwart nicht mehr wegzudenken. Um den ständig wachsenden Bedürfnissen hinsichtlich Speicherkapazität und Bandbreite gerecht zu werden, kommt es seit ca. 10 Jahren verstärkt zum Einsatz von Peer-to-Peer-Netzwerken (P2P). Der Umstieg von einer zentralen, Server-gesteuerten Netzstruktur hin zu einer Struktur mit vielen gleichberechtigten Netz-Teilnehmern verringert dabei die Kosten und den Verwaltungsaufwand und erhöht die Zuverlässigkeit des Netzwerkes.

Da die Netzwerk-Infrastruktur weiterhin permanent ausgebaut wird und sich so die Anzahl der Netzzugangsknoten auch in Zukunft noch erhöhen wird, sind auch die Netzbetreiber an einer Organisation dieser Zugangsknoten in einem P2P-Netzwerk interessiert, welches kostengünstig und robust ist. Bei der Bereitstellung des Netzzugangs werden auf jedem einzelnen Knoten eine Vielzahl von Datensätzen gespeichert, wie z.B. Log-Dateien und statistische Daten zu Netzwerkparametern und Nutzerinformationen, die für einen ordnungsgemäßen Betrieb unerlässlich sind. Bei einem Ausfall des Netzzugangsknotens geht ein Großteil dieser Daten verloren, da der Festwertspeicher bauartbedingt eine sehr begrenzte Speicherkapazität aufweist. Für die Wiederaufnahme des Betriebes werden diese Daten jedoch dringend benötigt.

Abhilfe schafft hier das Prinzip der dedizierten Speicherung im P2P-Netzwerk: die relevanten Daten werden auf andere Knoten verteilt und gespeichert. Um die Verfügbarkeit der Datensätze nach einem Geräteausfall weiter zu erhöhen und dabei die redundante Datenlast, die auf dem P2P-Netzwerk lastet, so gering wie möglich zu halten, bietet sich der Einsatz von Erasure Resilient Codes an. Durch sie ist es möglich, auch aus verlustbehafteten Daten, die originalen Datensätze

wiederherzustellen, da mit einem Ausfall weiterer Netzzugangsknoten jederzeit zu rechnen ist.

1.2 Aufbau der Bachelorarbeit

Zu Beginn dieser Bachelorarbeit erfolgt eine Gegenüberstellung der beiden Konzepte der verteilten Speicherung, welche durch Replikation oder durch Codierung mit Hilfe der Erasure Resilient Codes erfolgen kann. Um Erasure Resilient Codes einsetzen zu können, werden im Vorfeld einige grundlegende Kenngrößen und Grundlagen aus der Codierungstheorie vorgestellt. Anschließend werden einige ausgewählte Codierungsformen hinsichtlich ihrer Eigenschaften und Leistungsvermögen betrachtet, um eine Abschätzung zu tätigen, welcher für die Verwendung als Erasure Resilient Code in Frage kommen würde. Dabei handelt es sich ausschließlich um fehlerkorrigierende Codes, die dem Fachbereich der Kanalcodierung aus der Nachrichtentechnik entstammen. Aufgrund seiner Eigenschaften, fällt nun die Wahl auf Verwendung des Reed-Solomon-Codes als Erasure Resilient Code, der im Folgenden in seinen Eigenschaften näher betrachtet wird. Ebenso erfolgt eine Analyse der Algorithmen zum En- und Decoding auf Grundlage des Divisionsverfahrens. Die Implementierung dieser Algorithmen erfolgt in C++ unter der Entwicklungsumgebung Qt und wird nun Schritt für Schritt beschrieben. Eine abschließende Betrachtung der Implementierung hinsichtlich Funktionalität und Komplexität findet ebenso in diesem Kapitel statt. Das letzte Kapitel gibt letztendlich eine abschließende Bewertung und einen Ausblick zur Verwendung der Erasure Resilient Codes wieder.

2 Grundlagen und Konzept der verteilten Speicherung

2.1 Kriterien zur verteilten Speicherung in P2P-Netzwerken

Die verteilte Speicherung hat das Ziel, die Verfügbarkeit der ursprünglichen Daten zu erhöhen. Dies kann zum einen durch das komplette Kopieren der Daten oder aber durch das Teilen der Daten in mehrere Teilstücke, deren verschränkte Codierung und anschließende Verteilung auf verschiedene Speichermedien erfolgen. Bei beiden Verfahren ist der maßgebende Faktor, die Verfügbarkeit des einzelnen Speichersystems und wird im folgendem mit P_K bezeichnet. Dieser besteht beispielsweise bei Festplattenverbundsystemen durch die vom Hersteller gegebene mittlere Ausfallzeit (MTBF) einer jeden Platte und der Zeit, in der die defekte Festplatte gewechselt wird (MTTR). Auch bei P2P-Netzwerken wird ein jeder Knoten als unabhängig betrachtet und die Gesamtverfügbarkeit des Netzwerkes ergibt sich durch das Vorhandensein der einzelnen Knoten im Netzwerk, also ob ein Knoten gerade Online ist oder nicht. Die Verfügbarkeit eines Knotens hängt dabei von all seinen Komponenten, also Speicher-/Kommunikations- und Verarbeitungssysteme, ab. Die Systemverfügbarkeit P_K muss damit also als eine feste, gegebene Größe angesehen werden, die sich nur durch Austausch zuverlässigerer Hardware erhöhen lässt. Ein $P_K = 1$, bedeutet damit eine Verfügbarkeit von 100%, welches zwar sehr erstrebenswert ist, aber in der Praxis nicht erreicht werden kann.

Mit den Verfahren der verteilten Speicherung lässt sich aber dennoch die Datenverfügbarkeit, im folgenden P_F für die Verfügbarkeit der Datei bezeichnet, erhöhen. Die Erhöhung hat aber zum Nachteil, dass eine gewisse Redundanz an Daten auftritt, die zu speichernde Datenmenge wird also erhöht und kann als Kosten für eine höhere Verfügbarkeit angesehen werden. Um diesen Mehraufwand an Speicherkapazität auszudrücken, wird an dieser Stelle der storage overhead factor

bzw. stretch factor S eingeführt, der das Verhältnis von ursprünglicher Datenmenge zur verteilten Datenmenge ausdrückt. Ein Faktor $S = 1$ bedeutet die Verdoppelung der ursprünglichen Datenmenge und entsteht beispielsweise durch das Kopieren der Daten vom ursprünglichen Knoten auf einen anderen. Da Speicherplatz und Bandbreite des Netzwerkes nicht in beliebigem Maße zur Verfügung stehen, gilt es, den Faktor S so klein wie möglich, aber eben so groß wie nötig zu halten. Der Faktor S lässt sich durch eine geeignete Wahl des Verfahrens zur verteilten Speicherung beeinflussen.

Tabelle 2.1 enthält zusammenfassend alle grundlegenden Kenngrößen zur Klassifizierung der verteilten Speicherung.

Parameter	Bedeutung
P_K	Mittlere Verfügbarkeit des einzelnen Knotens
P_F	Verfügbarkeit der Datei
S	storage overhead bzw. stretch factor

Tabelle 2.1: Grundlegende Kenngrößen der verteilten Speicherung

2.2 Speicherung mittels Replikation

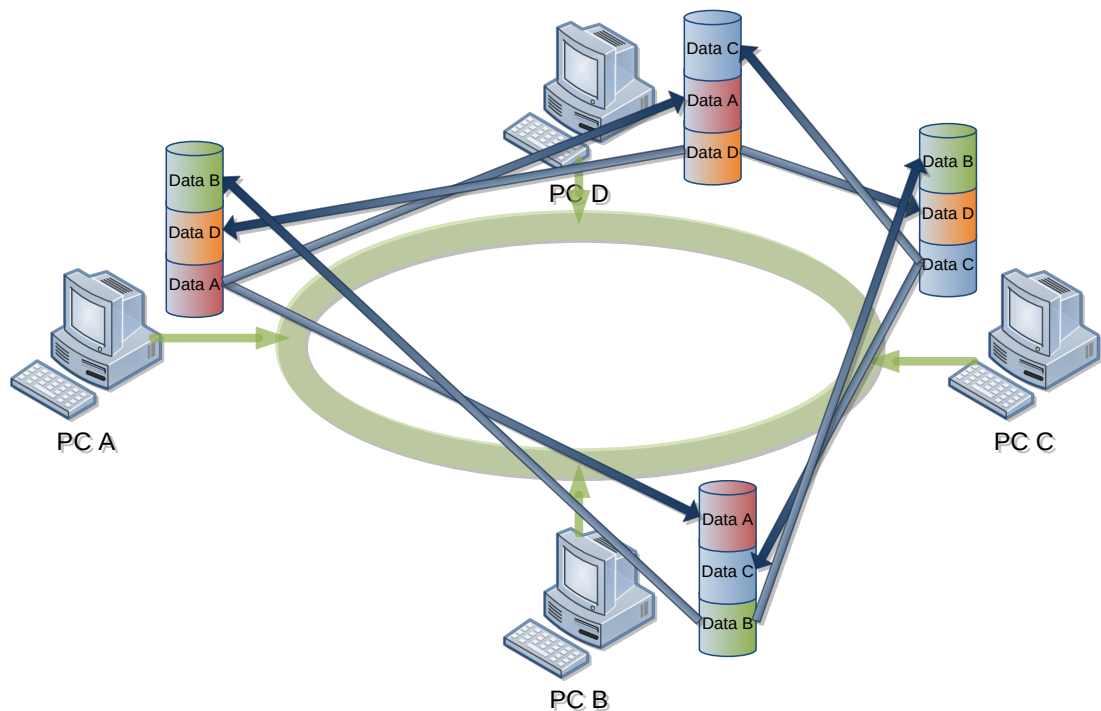


Abbildung 2.1: Grundprinzip der verteilten Speicherung mittels Replikation ($S = 2$)

Das einfachste Verfahren, die Verfügbarkeit der Daten durch verteilte Speicherung zu erhöhen, ist die Replikation. Dabei wird die Datei als Ganzes vom ursprünglichen Knoten auf andere Knoten kopiert. Der Faktor S gibt dabei die Anzahl an, wie oft die ursprüngliche Datei kopiert und auf verschiedene Knoten verteilt wurde. Nach Ausfall des Knotens muss dieser zur Wiederinbetriebnahme nur einen aus S funktionierenden Knoten im Netzwerk finden, und die ehemals auf ihm gespeicherten Daten auf sich übertragen. Das Grundprinzip ist der Abbildung 2.1 zu entnehmen. Die Verfügbarkeit der auf diese Weise verteilten Datei lässt sich mit folgender Formel vorhersagen:

$$P_F = \sum_{i=1}^S \binom{S}{i} P_K^i (1 - P_K)^{S-i} \quad (2.1)$$

Somit hängt die Verfügbarkeit einer Datei nach erfolgter Replikation nur von zwei Faktoren ab:

- P_K : Knotenverfügbarkeit
- S : Anzahl der verteilten Kopien im Netzwerk

Mittels eines Rechenbeispiels, soll die Erhöhung der Datenverfügbarkeit verdeutlicht werden. In einem fiktiven P2P-Netzwerk mit vielen Knoten erfolgt die verteilte Speicherung mit folgenden Kenngrößen:

- $P_K = 0.9$, d.h., die mittlere Verfügbarkeit des Knotens beträgt 90%
- $S = 2$, jeder Knoten erstellt zwei Kopien seiner Datei und verteilt sie an zwei andere Knoten

Danach ergibt sich die Datenverfügbarkeit:

$$P_F = \sum_{i=1}^2 \binom{2}{i} 0.9^i (1 - 0.9)^{2-i} = \underline{\underline{0.99}} \quad (2.2)$$

Daraus wird ersichtlich, dass durch das Verteilen von nur zwei Kopien der Datei im Peer-to-Peer-Netzwerk, es dem Knoten zu 99% gelingen wird, die Datei von einem anderen Knoten wiederzubeschaffen.

Tabelle 2.2 zeigt weitere Datenverfügbarkeiten in Abhängigkeit von P_K und Faktor S .

S	P_F bei $P_K = 70\%$	P_F bei $P_K = 80\%$	P_F bei $P_K = 90\%$
1	70	80	90
2	91	96	99
3	97,3	99,2	99,9
4	99,19	99,84	99,99
5	99,757	99,968	99,999

Tabelle 2.2: Datenverfügbarkeit (in %) im Falle der Replikation

2.3 Speicherung durch Verwendung von Erasure Resilient Coding

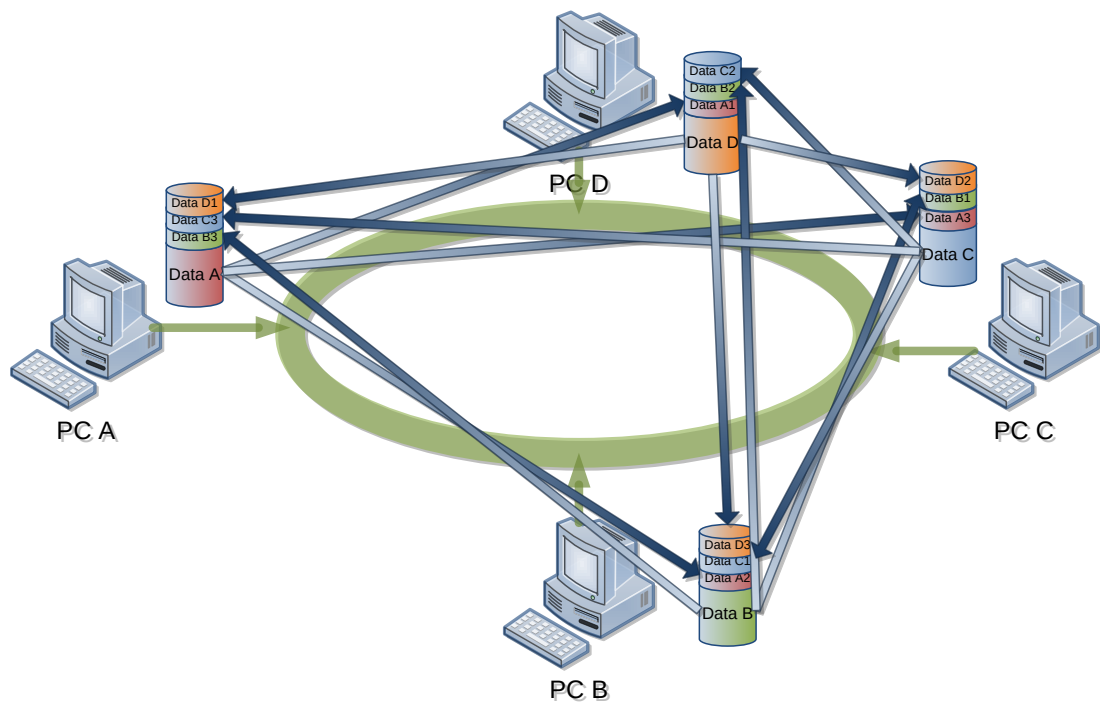


Abbildung 2.2: Grundprinzip der verteilten Speicherung mittels Erasure Resilient Codes

Bei Verwendung der Datenspeicherung mit Hilfe des Erasure Resilient Codings, müssen die ursprünglichen Daten vor der verteilten Speicherung aufbereitet werden. Dazu wird im ersten Schritt die Datei vom ausgehenden Knoten in m gleichgroße Blöcke geteilt. Anschließend erfolgt die Codierung dieser Datenblöcke mit Hilfe der Algorithmen der Erasure Resilient Codes. Dabei entstehen n Datenblöcke gleicher Größe, wobei gilt: $n > m$, die durch das Kodieren untereinander in Abhängigkeit stehen. Diese werden nun im Netzwerk jeweils auf einen unterschiedlichen Knoten verteilt. Der Vorteil des Erasure Resilient Codings besteht nun darin, dass der entsprechende Knoten zur Wiederherstellung seiner ursprünglichen Daten lediglich m aus n Datenblöcke finden und transferieren muss. Nach dem

Decodieren der m beliebigen Datenblöcke entstehen dann die m Ursprünglichen, aus denen dann die vollständige Datei durch Zusammensetzen in der richtigen Reihenfolge wieder rekonstruiert wird. Der Ablauf dieses Verfahrens der verteilten Speicherung kann der Abbildung 2.3 entnommen werden. Der Mehraufwand

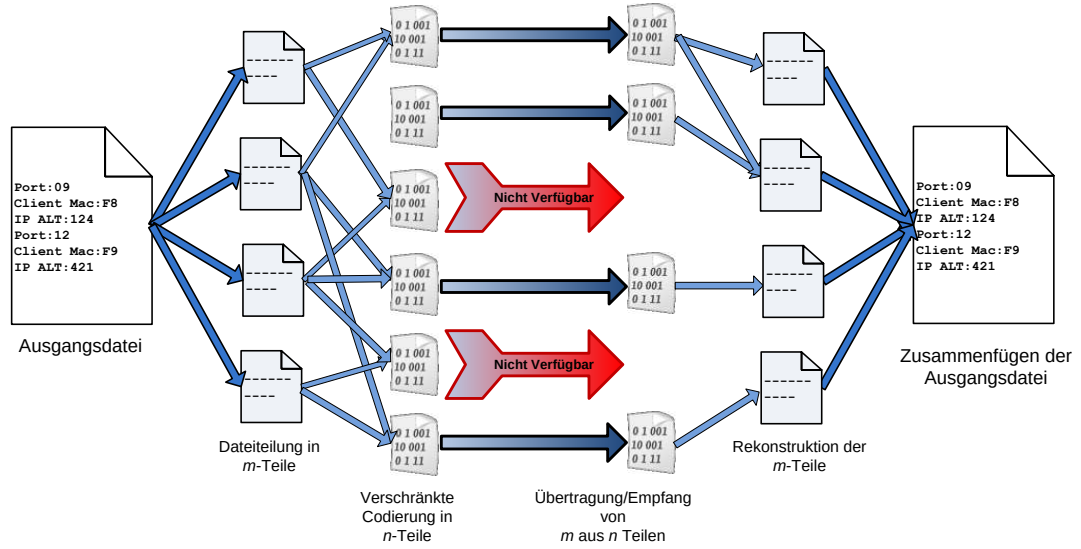


Abbildung 2.3: Grundprinzip der Codierung mittels Erasure Resilient Codes

an Speicher, der durch die redundanten Datenblöcke $n - m$ entsteht, lässt sich auch hier mit dem Faktor S verdeutlichen:

$$S = \frac{n}{m} \quad (2.3)$$

Die Verfügbarkeit P_F einer Datei, die mit Hilfe des Erasure Resilient Codings im Netzwerk verteilt wurde, lässt sich mit folgender Formel vorhersagen:

$$P_F = \sum_{i=m}^n \binom{n}{i} P_K^i (1 - P_K)^{n-i} \quad (2.4)$$

Die Datenverfügbarkeit hängt somit von drei Kenngrößen ab:

- m : Anzahl der Datenblöcke, in die die Datei geteilt wurde
- n : Anzahl der Datenblöcke, die nach dem Encoding entstanden sind und zum Speichern verteilt werden
- P_K : Knotenverfügbarkeit

Durch Variation der Variablen m und n ergeben sich dann unterschiedliche Datenverfügbarkeiten, wie in Tabelle 2.3 deutlich wird. Anhand dieser wird auch ersichtlich, dass sich die Verfügbarkeit mit der steigenden Anzahl der verteilten

Datenblöcke n bei konstantem Faktor S erhöht. Ebenso auffällig: ist das $n : m$ -Verhältnis sehr klein, also der Faktor S sehr gering, dann sinkt die Verfügbarkeit der Daten gegebenenfalls unter die der Knoten, wie im Falle von $m = 16$, $n = 20$, $S = 1,25$ zu erkennen ist.

m	n	S	P_F bei $P_K = 70 \%$	P_F bei $P_K = 80 \%$	P_F bei $P_K = 90 \%$
2	4	2	91,63	97,28	99,63
4	8	2	94,203235	98,95936	99,956835
8	12	1,5	72,365547	92,744445	99,567066
8	16	2	97,432647	99,852406	99,999408
16	20	1,25	23,750778	62,964826	95,682550
16	24	1,5	72,503703	96,382501	99,967874
16	32	2	99,4759	99,996728	99,999999
16	40	2,5	99,997872	99,999999	99,999999

Tabelle 2.3: Datenverfügbarkeit (in %) im Falle des Erasure Resilient Codings

2.4 Replikation vs. Erasure Resilient Coding

Grundsätzlich führen beide Verfahren der verteilten Speicherung zu einer Erhöhung der Verfügbarkeit der Daten nach einem Knotenausfall. Und unabhängig von der Wahl der verteilten Speicherung wird die Erhöhung der Datenverfügbarkeit durch einen Mehrverbrauch an Speicher erkaufte. Doch wie die Rechenbeispiele der Tabellen 2.2 und 2.3 zeigen, erreicht man mit Hilfe der ausgeklügelten Codierungsalgorithmen der Erasure Resilient Codes schon bei geringer Redundanz eine deutliche Erhöhung der Verfügbarkeit.

Will man beide Verfahren direkt miteinander vergleichen, so ist der Faktor S die gemeinsame Kenngröße, die den Mehrverbrauch an Speicher charakterisiert. Setzt man beispielsweise den Faktor auf $S = 2$, so ist für alle hier betrachteten Knotenverfügbarkeiten P_K eine deutliche Erhöhung der Datenverfügbarkeit durch den Einsatz der Erasure Resilient Codes zu sehen. Eine Erhöhung der Blockanzahl n läßt die Verfügbarkeit schnell gegen den angestrebten Wert von 100% konvergieren.

Nachteilig am Erasure Resilient Coding ist allerdings deren hohe Komplexität, hervorgerufen durch komplexe Rechenoperationen des En-/Decodings von vielen Datenblöcken.

Da es die Aufgabe der hier betrachteten Netzzugangsknoten ist, eine Vielzahl von Teilnehmern den Zugang zum Internet zu ermöglichen, ist eine schnelle Auf-

nahme des Betriebes äußerst erstrebenswert. In diesem Zusammenhang prägte sich in Industrie und im Dienstleistungsgewerbe der IT-Branche der Begriff der „five nines“ in Bezug auf die Verfügbarkeit [7]. Dies bedeutet, dass eine mittlere Verfügbarkeit von 99,999 %, bzw. in absoluter Schreibweise 0.99999, sichergestellt werden muss. Netzzugangsknoten sollten damit zu den hochverfügbaren Systemen gezählt werden, da sich die digitale Kommunikation zum Rückgrat eines jeden Wirtschaftssystems entwickelt hat.

3 Algorithmen zur Realisierung der Erasure Resilient Codes

3.1 Grundlagen der Codierungstheorie

Auch wenn bislang hier die Rede von Erasure Resilient Codes ist, so geht es doch in erster Linie um fehlerkorrigierende Codes (ECC). Eine Codierungsform, die in der Lage ist, in einem übertragenen Informationsblock einen Fehler zu erkennen und diesen zu korrigieren, ist auch in der Lage, eine Auslöschstelle wiederherzustellen. Vielmehr ist es die spezielle Eigenschaft der Erasure Resilient Codes, zwischen den eigentlichen Informationen und den zur Fehlerkorrektur benötigten redundanten Stellen ein möglichst optimales Verhältnis zu schaffen. Um eine Einschätzung darüber zu geben, sind im Vorfeld einige Kenngrößen und Gesetzmäßigkeiten zu betrachten.

Ein wichtiges Maß für die Fehlerkorrektur ist der Minimalabstand d_{min} . Er gibt die Anzahl der Stellen in einem Nachrichtencodewort an, in der sich zwei Wörter mindestens unterscheiden. Der Unterschied zwischen zwei Nachrichtenwörter lässt sich auch mittels der Hamming Distanz ausdrücken, und wird am einfachsten mittels einer Exklusiv-Oder-Verknüpfung (XOR) zwischen den beiden Wörtern und dem anschließendem Auszählen der Stellen, die ungleich Null sind, bestimmt. Die Hamming Distanz ist wie der Minimalabstand dabei nicht allein auf binäre Codewörter begrenzt, sondern kann auch auf Codewörtern aus Zahlen und Buchstaben angewendet werden. Folgendes Beispiel soll die Hamming Distanz ver-

deutlichen:

$$\begin{aligned}
& 1001 \\
\oplus \quad & 1011 = 0010 \rightarrow \textit{HammingDistanz} = 1 \\
\\
& 1234 \\
\oplus \quad & 1344 = 0110 \rightarrow \textit{HammingDistanz} = 2 \\
\\
& \textit{EINS} \\
\oplus \quad & \textit{ZWEI} = 1111 \rightarrow \textit{HammingDistanz} = 4
\end{aligned} \tag{3.1}$$

Wird ein Codewort ohne Redundanz übertragen, so beträgt $d_{min} = 1$, da sich die verschiedenen Codewörter immer um mindestens eine Stelle unterscheiden. Dies gilt allerdings nur unter der Voraussetzung, dass das Nachrichtencodewort eindeutig kodierbar ist, d.h., jedem Nachrichtenwort ist genau ein Code-Wort zugeordnet. Danach ist die Anzahl der korrigierbaren Fehler f_K wie folgt bestimmt:

$$f_K \leq \frac{d_{min} - 1}{2} \tag{3.2}$$

Wird die Ungleichung 3.2 mit Gleichheit erfüllt, so spricht man von einem „perfekten“ Code. Bei einem binären Code lässt sich die Anzahl der redundanten Stellen k bei Kenntnis der Nachrichtencodewortlänge n damit wie folgt berechnen:

$$2^k \geq \sum_{i=0}^{f_K} \binom{n}{i} \tag{3.3}$$

Der Minimalabstand d_{min} wird durch die Wahl der Parameter n und m maßgebend beeinflusst, und er wird durch die Singleton-Schranke begrenzt. Für einen linearen $(n, m, d_{min})_q$ -Code muss gelten [2]:

$$d_{min} \leq n - m + 1 \tag{3.4}$$

Wird die Singleton-Schranke mit Gleichheit erfüllt, so handelt es sich um einen MDS-Code (Maximum Distance Separable). Ein MDS-Code von der Struktur $(n, m, n - m + 1)_q$ hat dabei eine sehr wichtige Eigenschaft: Das Codewort kann eindeutig durch beliebige m aus n Symbole bestimmt werden. Dies wird bei der später noch erfolgenden Wahl eines entsprechenden Codes für das Erasure Resilient Coding von entscheidender Bedeutung sein.

Die entscheidenden Kenngrößen sind in Tabelle 3.1 zusammengefasst.

Parameter	Bedeutung
f_K	Anzahl der Symbole, die als Fehler erkannt/korrigiert werden
d_{min}	Minimalabstand
n	Anzahl der Symbole des Nachrichtencodewortes
m	Anzahl der Symbole des Quellencodewortes
k	Anzahl der redundanten Symbole, $n - m$
q	Symbolvorrat im Alphabet

Tabelle 3.1: Grundlegende Kenngrößen der Kodierungstheorie

3.2 Klassifizierung und Kurzbeschreibung von Erasure Resilient Codes

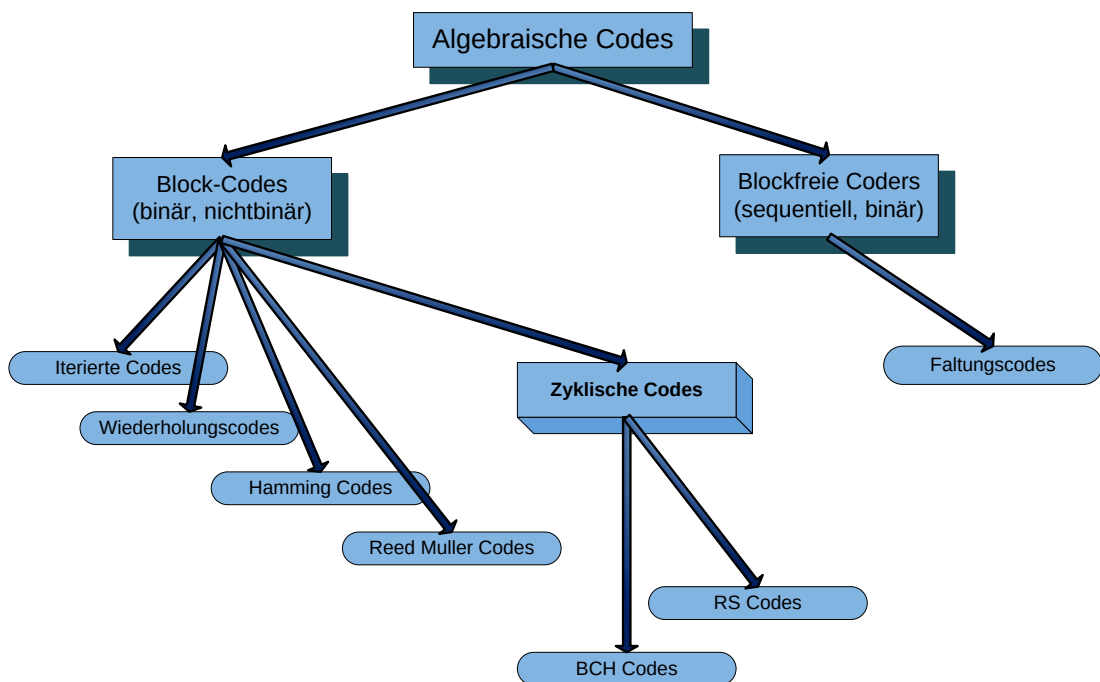


Abbildung 3.1: Klassifizierung von Error Correction Codes

Da anhand der Parameter n, m und k allein keine Aussage getroffen werden kann, welche Codierungsform die Geeignete sei, um das Problem der verteilten Speicherung zu lösen, soll im Folgenden eine kurze Übersicht über die verschiedenen Codierungsfamilien sowie eine Kurzbeschreibung der wichtigsten Code-Arten mit ihren Eigenschaften gegeben werden.

Dabei hängt die Wahl des entsprechenden Codes nicht allein von seinen Leistungsvermögen ab. Auch die technische Umsetzbarkeit und Realisierbarkeit spielt eine

wichtige Rolle. Ebenso fällt die Art der zu codierenden Daten schwer ins Gewicht, ebenso in welcher Art und Weise die codierten Daten übertragen werden.

Eine in der Praxis häufig verwendete und sehr bedeutsame Codegruppe, ist die Gruppe der Blockcodes. Sie ist dadurch gekennzeichnet, dass die verwendeten Codewörter eines Alphabets eine konstante Länge besitzen. Die Codewörter werden dabei durch Verknüpfung und Rechnungen gebildet, die innerhalb einer Gruppe als algebraischer Struktur stattfinden. Werden die ursprünglichen Quellencodewörter uncodiert, denen durch Codierung entstandenen Prüfzeichen vorangestellt, gespeichert oder übertragen, so spricht man von einem Systematischen Blockcode, wie er als Schema in Abb. 3.2 zu sehen ist.

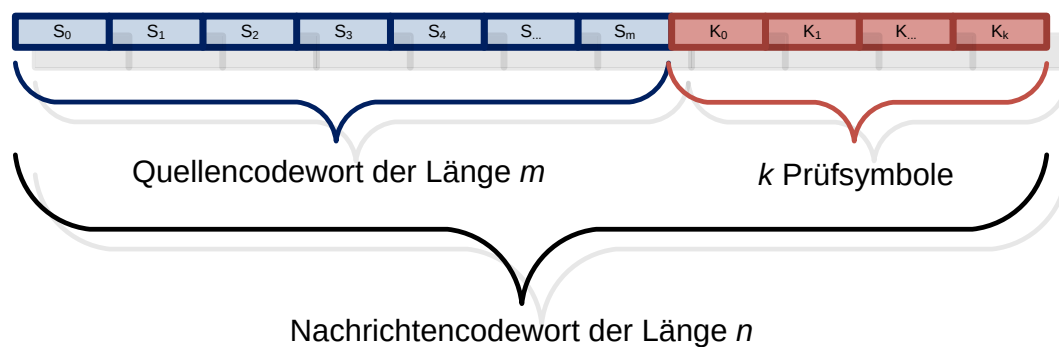


Abbildung 3.2: Schema des Systematischen Blockcodes

Iterierte Codes: Zu ihnen gehören unter anderem die Parity Check Codes, die meist für die interne Datenübertragung in Rechnersysteme bzw. zum Speichern von Daten auf lokalen Datenträgern genutzt werden. Dabei wird dem Quellencodewort im einfachsten Fall ein einziges Prüfbit hinzugefügt, welches die Summe aller binären Einsen $\text{mod} 2$ des Quellencodewortes annimmt (even parity). Mit der Erhöhung des Minimalabstandes $d_{\min}$ von eins auf zwei, ist es nun möglich, genau einen Fehler im übertragenem Nachrichtencodewort festzustellen. Eine Fehlerlokalisierung bzw. -korrektur ist damit allerdings nicht möglich.

Wiederholungs_codes: Die Fähigkeit, Fehler zu erkennen und im Bedarfsfall zu korrigieren, erlangen die Wiederholungs_codes, auch als repetition codes bekannt, durch wiederholtes Senden ein und des selben Quellencodesymbols. Tritt eine Verfälschung oder der Verlust eines Nachrichtensymbols auf, so wird mittels Mehrheitsentscheid das vermeintlich richtige Symbol aus den anderen Symbolen des Codewortes ausgewählt. Mit diesem einfachen Korrekturverfahren ließe sich bei entsprechender Nachrichtencodewortlänge n eine fast fehlerfreie Übertragung/Speicherung von Daten erreichen, jedoch

ist die Redundanz der hinzugefügten Symbole k und damit der Faktor S sehr groß ($S \geq 3$).

Hamming Codes: Diese Codes eignen sich besonders zur Ein-Fehlerkorrektur. Durch Hinzufügen von Prüfbits zum binären Quellencodewort wird der Minimalabstand auf $d_{min} = 3$ erhöht. Dadurch wird es möglich, ein fehlerhaftes Bit im Nachrichtencodewort zu erkennen und zu korrigieren. Durch Einbringen eines weiteren Prüfbits kann zwar die Anzahl der erkennbaren Fehler auf zwei erhöht werden, aber nur ein fehlerhaftes Bit kann rekonstruiert werden.

Reed-Muller-Codes: Als Erweiterung des Hamming-Codes, ist dieser binäre Code, meist in Kurzform RM-Code genannt, in der Lage, auch Mehrfachfehler zu erkennen und zu korrigieren. Durch geeignete Codier- und Decodieralgorithmen lassen sich auch Codes erzeugen, die ähnlich leistungsfähig wie BCH-Codes sind. Jedoch ist man hinsichtlich der Wahl der Parameter zur Code-Konstruktion etwas eingeschränkt.

Zyklische Codes: Diese Code-Gruppe besitzt einen ganz besonderen Stellenwert, da sie in vielen praktischen Anwendungsfällen zum Zuge kommen. Neben der Erkennung von Einzelfehlern und deren Korrektur, sind sie auch hervorragend geeignet, Bündelfehler (burst errors) zu erkennen und zu rekonstruieren. Eine Eigenschaft der Zyklischen Codes, zu deren Namensgebung es damit auch führte, ist es, dass durch Verschiebung eines Code-Wortes, wiederum ein neues Code-Wort entsteht. Die Codierung erfolgt meist mit Hilfe von Polynomen, wodurch sich diese Codierungsform sehr gut durch rückgekoppelte Schieberegister (LFSR) hardware-technisch realisieren lässt. Die bekanntesten Vertreter dieser Gruppe sind der binäre BCH-Code, ernannt nach seinen Entdeckern R. C. Bose, D. K. Ray-Chaudhuri und A. Hocquenghem, und der Symbol-orientierte Reed-Solomon-Code (kurz: RS), welcher von I. Reed und G. Solomon entdeckt wurde.

Die Blockfreien Codes, Faltungscodes oder auch convolutional Codes, wie sie auch bezeichnet werden, arbeiten im Unterschied zu den Blockcodes mit einem kontinuierlichen Datenstrom. Dabei wird die Redundanz als Ergebnis der Faltung der Quellencodewörter kontinuierlich in das Nachrichtencodewort eingefügt. Blockfreie Codes eignen sich hervorragend zur Korrektur zufällig verteilter Einzelfehler und werden meist mit anderen Codes mittels Codeverkettung eingesetzt, um so die Leistungsfähigkeit zu steigern.

Die hier genannten Codes stehen natürlich nur exemplarisch für eine Vielzahl von

Codierungsformen, und die Übersicht erhebt keinerlei Anspruch auf Vollständigkeit. Darüber hinaus werden in der praktischen Realisierung der Fehlererkennung und -korrektur, verschiedene Codes seriell oder parallel miteinander verkettet (Interleaving), um die günstigen Eigenschaften der jeweiligen Codes zu kombinieren und den so kombinierten neuen Code für neue Anwendungsbereiche anwendbar zu machen.

3.3 Reed-Solomon Codes

3.3.1 Grundlagen und Kenngrößen

Nach der Betrachtung der verschiedenen Codierungsformen und ihren Eigenschaften in dem vorangegangenen Abschnitt, fällt nun die Entscheidung, den Reed-Solomon-Code als Erasure Resilient Code zu benutzen.

Die Reed-Solomon-Codes wurden bereits 1960 durch Irving Reed und Gustave Solomon entwickelt. Doch durch den Mangel an entsprechend leistungsfähiger Rechentechnik, und dem Umstand, dass eine Implementierung der Algorithmen in Hardware (ASIC) zur damaligen Zeit sehr aufwändig und kostenintensiv war, erfolgte die erste kommerzielle, in Massen verwendete Nutzung der RS-Codes erst 1982 als Fehlerkorrekturverfahren in der CD. Aufgrund der günstigen und vielfältigen Eigenschaften finden die Reed-Solomon-Codes heute eine weite Verbreitung und kommen unter anderem bei der DVD, Satellitenkommunikation, Netzwerkkommunikation (DSL, WiMAX) und der Video-Übertragung DVB zur Anwendung [4].

Wie bereits erwähnt, werden mittels der Reed-Solomon-Codes Symbole codiert, die einem Alphabet A angehören und über einen algebraischen Körper $GF(q)$ gebildet werden. Innerhalb dieses Körpers werden alle Rechenoperationen durchgeführt. Die Anzahl der Symbole wird dabei üblicherweise als Potenz zur Basis 2 angegeben, z.Bsp. $q = 2^8$ für $q = 256$ Symbole. Andere Größenordnungen sind aber ebenso realisierbar. Aus der Anzahl der Symbole q und dem daraus resultierenden Körper, ergibt sich auch die maximal zulässige Blocklänge n :

$$n \leq 2^{k_1} - 1 \tag{3.5}$$

k_1 steht dabei für die Anzahl der Bit-Stellen eines Symbols. So gilt beispielsweise bei einer Codierung von Oktalzahlen: $n \leq 7$ über $GF(2^3)$, bei Hexadezimal-

Zeichen: $n \leq 15$ über $GF(2^4)$ und bei Codierung über Byte-Symbole: $n \leq 255$ über $GF(2^8)$.

Eine wichtige Eigenschaft der Reed-Solomon-Codes ist, dass sie die Singleton-Schranke mit Gleichheit erfüllen:

$$d_{min} = n - m + 1 \quad (3.6)$$

Für die Anzahl der korrigierbaren Fehler im Nachrichtencodewort der Länge n bedeutet dies:

$$f_K = \frac{(n - m + 1) - 1}{2} = \frac{n - m}{2} = \frac{k}{2} \quad (3.7)$$

D.h., durch Einbringen von jeweils zwei Prüfsymbolen, kann jeweils ein Fehler korrigiert werden. Durch Erfüllung der Ungleichung der Singleton-Schranke mit Gleichheit, handelt es sich bei den Reed-Solomon-Codes um einen MDS-Code. Es werden also nur m aus n Symbole benötigt, um Auslöschstellen zu rekonstruieren. Sind also all jene Stellen bekannt, die bei der Übertragung/Speicherung verloren gegangen sind, und davon kann man bei der verteilten Speicherung ja ausgehen, dann erreicht dieser Code die doppelte Effektivität gegenüber der reinen Fehlerkorrektur. Für jedes eingebrachte redundante Symbol kann somit genau ein Symbol einer Auslöschstelle wiederhergestellt werden. Und genau diese Eigenschaft macht den Reed-Solomon-Code so geeignet für die verteilte Speicherung mittels Erasure Resilient Coding. Grundvoraussetzung ist jedoch, dass die nicht ausgelöschten Symbole fehlerfrei übertragen bzw. gespeichert werden. Allerdings verfügen Speichermedien und Netzwerkprotokolle bereits über geeignete Maßnahmen zur Fehlererkennung und -korrektur.

Zur Bezeichnung der Reed-Solomon-Codes findet man in der Fachliteratur diverse unterschiedliche Möglichkeiten. Eine der Gängigsten ist aber, wenn man von einem RS(n,m)-Code spricht. Eines der Standard-Codes der NASA zur Satellitenkommunikation ist beispielsweise ein RS(255,223)-Code. Daraus kann man die Länge des Nachrichtencodewortes n von 255 Symbolen und die Anzahl der Nutzsymbole m mit 223 Symbole direkt ablesen. Die Anzahl der redundanten Symbole beträgt dann $k = n - m = 32$, welches eine Fehlerkorrekturleistung von maximal 16 Symbolen, bzw. Auslöschkorrekturleistung von maximal 32 Symbolen bedeutet. Wird die maximale Blocklänge n nicht ausgenutzt, wie es bei der CD mit einem RS(28,24)-Code beispielsweise der Fall ist, spricht man von einem verkürzten Code. Angaben über den zur Bildung benutzten Körper werden dabei meist nur dann getätigt, wenn es nicht mit Hilfe eines $GF(2^8)$ -Körpers geschieht.

3.3.2 Algorithmus zum Encoding

Erstellen des erweiterten Galoiskörpers

Grundlage für alle Rechenoperationen während des Encoding und Decoding ist der endliche Erweiterungskörper $GF(q)$. Dieser enthält die Polynomreste, die aus der Polynomdivision des Modularpolynoms durch das Symbol in Potenzschreibweise über $GF(2)$ hervorgehen. An den Eigenschaften des Modularpolynoms $M(x)$ werden einige Bedingungen geknüpft, die die grundlegenden Eigenschaften des zu entwerfenden Codes bestimmen. Das Modularpolynom muss irreduzibel sein, d.h., es darf nicht in ein Produkt von Polynomen zerlegbar sein. Der Grad des Modularpolynoms $grad(M(x))$ bestimmt die binäre Blocklänge des Symbols aus dem Alphabet A über den zu erstellenden Erweiterungskörper $GF(2^{k_1})$ und damit auch die maximale Anzahl der verwendbaren Symbole:

$$k_1 = grad(M(x)) \quad (3.8)$$

Will man nun alle möglichen Symbole des Körpers $GF(2^{k_1})$ eindeutig codieren,

$grad(M(x))$	Modularpolynome $M(x)$
3	$x^3 + x + 1$
4	$x^4 + x + 1$
8	$x^8 + x^6 + x^5 + x^4 + 1$
16	$x^{16} + x^{12} + x^3 + x + 1$

Tabelle 3.2: Ausgewählte Modularpolynome

so muss das Modularpolynom auch die Eigenschaft eines primitiven Polynoms besitzen. Dadurch erreicht man die maximale Zykluslänge von $(2^{k_1} - 1)$ der Polynomreste im Erweiterungskörper $GF(2^{k_1})$. Hat das Modularpolynom diese Eigenschaft nicht, so ist der Zyklus der Polynomreste kürzer als $(2^{k_1} - 1)$, d.h., sie wiederholen sich, wodurch nicht mehr alle möglichen Symbole eindeutig codierbar sind. Die Modularpolynome entsprechenden Grades kann man dabei als gegeben voraussetzen und aus entsprechenden Tabellen ablesen [9]. Eine kleine Auswahl der Gebräuchlichsten sind in Tabelle 3.2 zu finden.

Der erste Schritt zum Aufbau des Erweiterungskörpers besteht nun darin, die Symbole des Alphabets A , welche für gewöhnlich die Wertigkeit 0 bis $(2^{k_1} - 1)$ besitzen, umzubenennen. Die „0“ wird zum Nullelement, die „1“ zu α^0 , und das letzte Symbol $(2^{k_1} - 1)$ wird zum Element $\alpha^{2^{k_1}-2}$ des Galoiskörpers $GF(2^{k_1})$. Als nächstes werden die Polynomreste mittels der Polynomdivision über $GF(2)$ bestimmt, wobei das Element aus dem Galoiskörper $GF(2^{k_1})$ den Dividenden und

das Modularpolynom den Divisor darstellt.

Beispiel: Es ist das Restpolynom zu berechnen, welches dem Element α^5 über $GF(2^3)$ zugehörig ist, durch Verwendung des Modularpolynoms

$$M(x) = x^3 + x^2 + 1:$$

$$\begin{array}{r}
 \alpha^5 \qquad \qquad \qquad : \alpha^3 + \alpha^2 + 1 = \alpha^2 + \alpha + 1 \\
 \underline{\alpha^5 + \alpha^4} \qquad \qquad + \alpha^2 \\
 \alpha^4 \qquad \qquad + \alpha^2 \\
 \underline{\alpha^4 + \alpha^3} \qquad \qquad + \alpha \\
 \alpha^3 + \alpha^2 + \alpha \\
 \underline{\alpha^3 + \alpha^2} \qquad \qquad + 1 \\
 \underline{\alpha + 1}
 \end{array} \tag{3.9}$$

Die Koeffizienten der Polynomreste können dabei auch als Binärwerte interpretiert werden, deren Wertigkeit dann wiederum ein Element des Galoiskörpers $GF(2^{k_1})$ darstellt. Die Potenzschreibweise der Symbole führt bei den noch kom-

Elemente $GF(2^3)$	Polynomreste	Binäre Koeffizienten		
<i>Nullelement</i>		0	0	0
α^0		1	0	0
α^1	α		0	1
α^2	α^2		1	0
α^3	$\alpha^2 + 1$		1	0
α^4	$\alpha^2 + \alpha + 1$		1	1
α^5	$\alpha + 1$		0	1
α^6	$\alpha^2 + \alpha$		1	1

Tabelle 3.3: Erweiterungskörper $GF(2^3)$

menden Berechnungen des Encoding/Decoding zu vielen Vereinfachungen der Rechenoperationen unter zu Hilfenahme der Potenzregeln. Aus der Multiplikation zweier Symbole, wird die Addition ihrer Exponenten *Modulo*($2^{k_1} - 1$), und aus der Addition/Subtraktion zweier Symbole wird die XOR-Verknüpfung ihrer zugehörigen Polynomreste des Erweiterungskörpers.

Berechnung des Generatorpolynoms

Die Generierung des Nachrichtencodewortes wird, wie bei allen zyklischen Codes, mit einem Generatorpolynom $g(x)$ durchgeführt. Dabei ist das Generatorpolynom im allgemeinen ein Produkt aus Minimalpolynomen $m_i(x)$ über $GF(2^{k_1})$. Die

Nullstelle eines Minimalpolynoms wird mit $m_i(x) = (x + \alpha^i)$ beschrieben. Nachrichtencodewort, als auch Generatorpolynom, haben dann ebenfalls als Nullstelle aufeinanderfolgende Potenzen von α . Das Generatorpolynom lässt sich damit wie folgt konstruieren:

$$g(x) = \prod_{i=\mu}^{\mu+d_{min}-2} (x + \alpha^i) \quad (3.10)$$

Da für MDS-Codes $d_{min} = k+1$ gilt, lässt sich die Bildung des Generatorpolynoms auch wie folgt vornehmen:

$$g(x) = \prod_{i=\mu}^{\mu+k-1} (x + \alpha^i) \quad (3.11)$$

Der Parameter μ ist dabei lediglich ein Start-Wert und kann beliebig gewählt werden. In allen Beispielen, als auch in der Implementierung wird ein $\mu = 1$ verwendet. Aus der Bildungsformel für das Generatorpolynom wird ebenso ersichtlich, dass der Grad des Generatorpolynoms $grad(g(x))$ der Anzahl der redundanten Symbole k entspricht:

$$grad(g(x)) = k \quad (3.12)$$

Anhand eines Beispiels soll hier die Bildung eines Generatorpolynoms nochmals verdeutlicht werden. Außerdem werden damit die neuen Rechenregeln innerhalb des erweiterten Körpers $GF(2^{k_1})$ gezeigt.

Beispiel: Es ist das Generatorpolynom für einen RS-Code zu erzeugen, die zu codierenden Symbole sind Elemente des Körpers $GF(2^3)$ gemäß Tabelle 3.3 und der Minimalabstand ist mit $d_{min} = 3$ gegeben:

$$\begin{aligned} g(x) &= \prod_{i=1}^2 (x + \alpha^i) \\ g(x) &= (x + \alpha^1)(x + \alpha^2) \\ &= x^2 + \alpha^2 x + \alpha x + \alpha^{(1+2) \bmod 7} \\ &= x^2 + x(\alpha^2 + \alpha) + \alpha^3 \\ &\quad \alpha^2 + \alpha = \alpha^2 \oplus \alpha \\ &\quad \alpha^2 \oplus \alpha = 100 \oplus 010 = 110 = \alpha^4 \\ g(x) &= \underline{\underline{x^2 + \alpha^4 x + \alpha^3}} \end{aligned} \quad (3.13)$$

Encoding

Der eigentliche Vorgang des Encoding generiert auf Grundlage der Quellsymbole die redundanten Symbole, die zur Fehler- und Auslöschkorrektur benötigt werden. Für diesen Vorgang gibt es verschiedene Möglichkeiten und Ansätze. Mit folgenden Verfahren ist die Codierung möglich:

- mit Hilfe einer Generator-/Kontrollmatrix
- Multiplikationsverfahren
- mit Hilfe der Diskreten Fourier-Transformation
- Divisionsverfahren

Alle folgenden Betrachtungen und die Implementierung werden sich einzig auf das Divisionsverfahren beziehen, da es von großer praktischer Bedeutung ist, denn dadurch wird ein systematischer Blockcode erzeugt, d.h., die uncodierten Quellsymbole bilden bereits einen Teil des Nachrichtencodewortes. Ein weiterer Vorteil des Divisionsverfahren ist die relativ einfache Realisierung der Codierung in als ASIC durch den Einsatz der LFSR.

Im ersten Schritt des Encodings wird das Quellencodewort $a(x)$ um die Anzahl der redundanten Symbole k , welches dem Grad des Generatorpolynoms $\text{grad}(g(x))$ entspricht, verschoben. Jedes Symbol nimmt dabei eine um k erhöhte Wertigkeit an. Nun erfolgt die Polynomdivision des Quellencodewortes durch das Generatorpolynom. Das daraus resultierende Restpolynom enthält genau k Koeffizienten. Das Nachrichtencodewort $n(x)$ wird anschließend aus dem uncodierten Quellencodewort und den Koeffizienten des Restpolynoms gebildet und entspricht der vorher definierten Blocklänge n .

Auf Grundlage der Beispiele 3.9 und 3.13 soll der letzte Schritt des Encodings verdeutlicht werden.

Beispiel: Das Quellencodewort $a = (\alpha^4\alpha^6\alpha^1)$ ist mit dem Generatorpolynom $g(x) = x^2 + \alpha^4x + \alpha^3$ über den Erweiterungskörper $GF(2^3)$ zu codieren. Damit ergeben sich folgende Kenngrößen des Codes: $m = 3$, $k = \text{grad}(g(x)) =$

2, $n = m + k = 5$. Es handelt sich also um einen RS(5,3)-Code.

$$a(x) = \alpha^4 x^2 + \alpha^6 x + \alpha^1$$

$$\text{Es gilt : } a(x)x^k = a(x)x^2 = \alpha^4 x^4 + \alpha^6 x^3 + \alpha^1 x^2$$

$$\begin{array}{rcl}
 \alpha^4 x^4 + \alpha^6 x^3 + \alpha^1 x^2 & : x^2 + \alpha^4 x + \alpha^3 = & \alpha^4 x^2 + \alpha^5 x + \alpha^5 \\
 \underline{\alpha^4 x^4 + \alpha^1 x^3 + \alpha^0 x^2} & & \\
 \alpha^5 x^3 + \alpha^3 x^2 & & \\
 \underline{\alpha^5 x^3 + \alpha^2 x^2 + \alpha^1 x} & & \\
 \alpha^5 x^2 + \alpha^1 x & & \\
 \underline{\alpha^5 x^2 + \alpha^2 x + \alpha^1} & & \\
 \underline{\underline{\alpha^4 x + \alpha^1}} & &
 \end{array} \tag{3.14}$$

$$n(x) = \alpha^4 x^4 + \alpha^6 x^3 + \alpha^1 x^2 + \alpha^4 x + \alpha^1$$

3.3.3 Algorithmus zum Decoding

Der Decoder hat nun im Rahmen des Erasure Resilient Coding die Aufgabe, verloren gegangene Symbole aus dem Empfangspolynom $b(x)$ zu rekonstruieren. Das wird ihm auch gelingen, sofern folgende Bedingungen erfüllt werden:

1. Mindestens m aus n Symbole müssen vorhanden sein.
2. Die vorhandenen Symbole müssen fehlerfrei übertragen/gespeichert worden sein.
3. Alle Symbole stehen in der richtigen Reihenfolge im Empfangspolynom.

Ist das Empfangspolynom zudem vollständig, d.h., es existieren keine Auslöschstellen, so ist eine Decodierung unnötig, da durch das Divisionsverfahren ein Systematischer Blockcode erzeugt wurde. Das Quellencodewort wird durch Streichen der redundanten Stellen im Empfangspolynom direkt wiederhergestellt. Dies gilt selbstverständlich auch dann, wenn nur die redundanten Symbole k von Auslöschungen betroffen sind.

Kommt es jedoch an den Positionen des Quellencodewortes zu Auslöschungen, so wird im ersten Schritt des Decodierens ein Auslöschvektor U angelegt, in dem die Wertigkeiten der ausgelöschten Symbole mit den äquivalenten Symbolen aus $GF(2^{k_1})$ mit steigender Potenz eingetragen werden.

Im zweiten Schritt werden die Fehlersyndrome s_j in der Anzahl der Auslöschstellen berechnet. Die Fehlersyndromberechnung dient dabei der Untersuchung der Nullstellenfolge im Empfangspolynom $b(x)$ und wird wie folgt berechnet:

$$s_j = b(x = \alpha^j) \quad (j=1,2,\dots,p: \text{Anzahl der Auslöschungen}) \quad (3.15)$$

Im nächsten Schritt erfolgt die Aufstellung eines Gleichungssystems zur Berechnung der Fehlerwerte y , die die tatsächlich Symbole an den Auslöschstellen repräsentieren. Die Gleichungen werden wie folgt aufgestellt:

$$s_j = \sum_{i=1}^p y_i x_i^j \quad p: \text{Anzahl der Auslöschstellen} \quad (3.16)$$

Die Werte für y werden nun durch Lösen des Gleichungssystems berechnet und an den Positionen der Auslöschstellen in das Empfangspolynom $b(x)$ eingesetzt. Damit ist das ursprüngliche Nachrichtencodewort $n(x)$ wiederhergestellt. Durch Streichen der k redundanten Symbole erhält man dann auch das ursprüngliche Quellencodewort. Alle dazu notwendigen Berechnungen finden auch hier wieder innerhalb des Erweiterungskörpers $GF(2^{k_1})$ statt.

Auch hier sollen die Beispiele 3.9, 3.13 und 3.14 weitergeführt werden, um den gesamten Decoding-Vorgang zu verdeutlichen.

Beispiel: Bedingt durch Auslöschungen wurde statt des Polynoms $b(x) = \alpha^4 x^4 + \alpha^6 x^3 + \alpha^1 x^2 + \alpha^4 x + \alpha^1$ das Polynom $b(x) = \alpha^4 x^4 + _ + _ + \alpha^4 x + \alpha^1$ interpretiert. Es kam also zur Auslöschung von zwei Symbolen, hier mit $_$ gekennzeichnet. Die Kenngrößen des Codes sind dem Decoder bekannt und

gegeben mit: $m = 3$, $k = 2$, $n = 5$.

$$\begin{aligned}
U &= (x_1, x_2) = (\alpha^2, \alpha^3) \\
s_1 &= b(x = \alpha^1) \\
b(\alpha^1) &= \alpha^4(\alpha^1)^4 + 0 + 0 + \alpha^4\alpha^1 + \alpha^1 \\
&= \alpha^4\alpha^4 + 0 + 0 + \alpha^4\alpha^1 + \alpha^1 \\
b(\alpha^1) &= \alpha^1 \oplus \alpha^5 \oplus \alpha^1 = \alpha^5 \\
s_1 &= \alpha^5 \\
s_2 &= b(x = \alpha^2) \\
b(\alpha^2) &= \alpha^4(\alpha^2)^4 + 0 + 0 + \alpha^4\alpha^2 + \alpha^1 \\
&= \alpha^4\alpha^1 + 0 + 0 + \alpha^4\alpha^2 + \alpha^1 \\
b(\alpha^2) &= \alpha^5 \oplus \alpha^6 \oplus \alpha^1 = 0 \\
s_2 &= 0 \\
s_1 &= y_1x_1 + y_2x_2 \\
s_2 &= y_1x_1^2 + y_2x_2^2 \\
\begin{pmatrix} \alpha^5 \\ 0 \end{pmatrix} &= \begin{pmatrix} \alpha^2 & \alpha^3 \\ \alpha^4 & \alpha^6 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \\
y_1 &= \alpha^1 \\
y_2 &= \alpha^6 \\
b_{Korr} &= \underline{\underline{(\alpha^4\alpha^6\alpha^1\alpha^4\alpha^1)}}
\end{aligned} \tag{3.17}$$

4 Implementierung und Verifikation

4.1 Die Entwicklungsumgebung Qt

Zur Implementierung der Algorithmen des Reed-Solomon-Codes wurde das Software-Entwicklungssystem Qt in der Version 4.5.2 unter Windows XP genutzt. Qt, welches von der norwegischen Firma Trolltech entwickelt wurde und aktuell durch Nokia vertrieben wird, ist dabei eine Sammlung von Klassenbibliotheken, die auf systemeigene C und C++ Bibliotheken aufbaut. Die Implementierung von Anwendungen in Ansi-C und C++ ist damit genauso möglich, wie mit jedem anderen Entwicklungssystem auch. Darüber hinaus stellt Qt dem Softwareentwickler mächtige Werkzeuge zur Verfügung, auf einfache und schnell zu realisierende Weise Anwendungen mit grafischer Benutzeroberfläche (GUI) in C++ zu entwickeln [6]. Dazu steht dem Entwickler das Tool „GUI Designer“ zur Verfügung, mit deren Hilfe via Drag & Drop Programmoberflächen erzeugt werden. Eine weitere Erleichterung stellt Qt's Konzept der „signals and slots“ dar, über die die Aktionen/Reaktionen der grafischen Objekte erfolgen.

Ein weiterer positiver Aspekt ist, dass Qt auch als Open Source Version zu beziehen ist und damit kostenfrei ist, sofern damit keine kommerzielle Software entwickelt und in Umlauf gebracht wird. Der Funktionsumfang von Qt ist dabei in keiner Weise eingeschränkt.

Aber ein wesentlicher Vorteil von Qt gegenüber anderen konkurrierenden Entwicklungssystemen ist dessen Plattformunabhängigkeit. Ist der Quelltext einmal entworfen und verifiziert, so kann er auf eine Vielzahl von Systemen portiert und kompiliert werden, ohne Änderungen an ihm vornehmen zu müssen. Dazu verfügt Qt über einen Präprozessor, genannt MOC (Meta Object Compiler), der den Quelltext in C++ Standardcode übersetzt. Dieser kann dann mit einem auf dem jeweiligen System gebräuchlichen Compiler, z.B. GCC, kompiliert werden. Die im Quelltext verwendeten GUI-Objekte werden dann an die entsprechenden Gegebenheiten des Betriebssystems angepasst. Das verringert den Entwicklungsaufwand und die Entwicklungszeit von Software und verschafft ihr ein weites

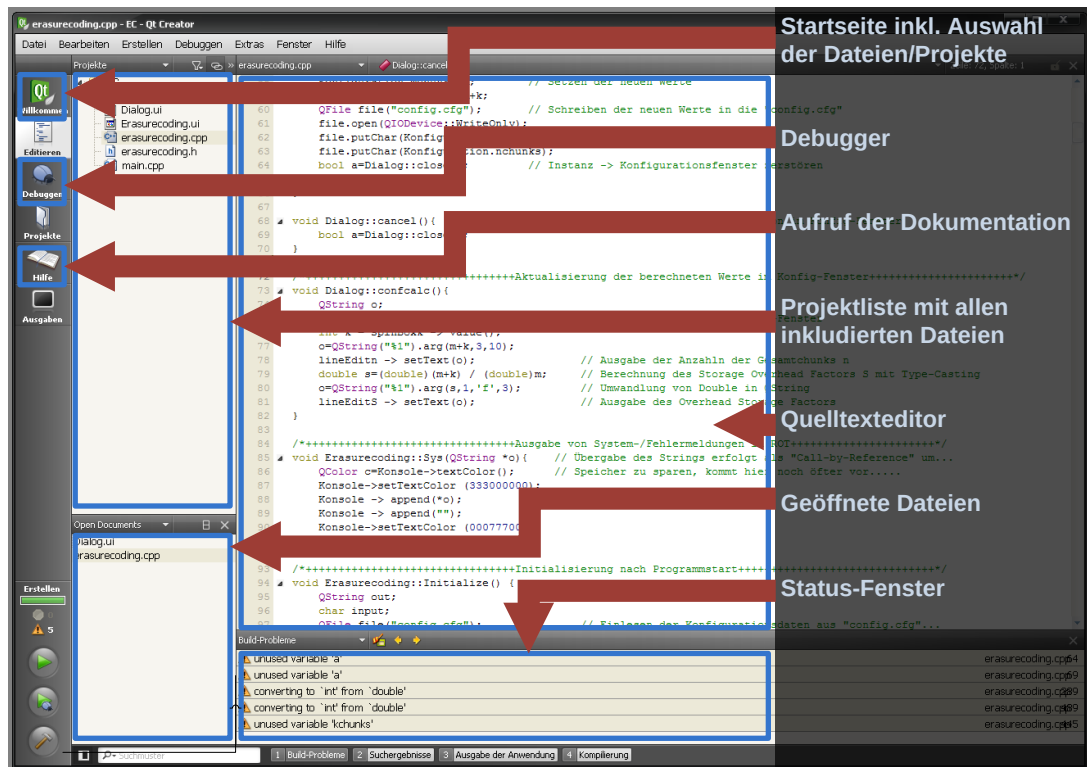


Abbildung 4.1: IDE von Qt V.4.5.2

Anwendungsspektrum auf verschiedenen Systemen. Aktuell lässt sich die mit Qt entwickelte Software auf folgende Systeme portieren [5]:

- Windows/Windows CE
- Mac OS X
- Linux/Embedded Linux
- diverse X11-Plattformen wie z.Bsp. Solaris, etc
- Nokia Symbian S60

Wie bei Entwicklungssystemen üblich, enthält auch Qt eine einfach zu bedienende integrierte Entwicklungsumgebung (IDE) mit den Grundkomponenten Quelltexteditor, Projekt-Datei-Liste, Liste der geöffneten Dateien und einem Statusfenster für Meldungen. Die umfangreiche Dokumentation zu den Qt spezifischen Befehlen, über 500 Klassen und 9000 Funktionen stehen seit der Version 4 zur Verfügung, kann jederzeit direkt aufgerufen werden. Der Aufbau der IDE kann der Abbildung 4.1 entnommen werden.

4.2 Das Hauptfenster: Bedienelemente und Funktionen

Wie die Verwendung von Qt als Entwicklungstool schon vermuten lässt, erfolgte die Implementierung unter einer grafischen Benutzeroberfläche und lässt sich vollständig mit der Maus bedienen. Das Hauptfenster, welches als Hauptklasse vom Typ `QMainWindow` definiert ist, stellt dabei alle Funktionen direkt zur Verfügung. Die Objekte und das Layout der Hauptklasse wurden dabei mit dem „GUI-Designer“ erstellt und in der Datei „Erasurecoding.ui“ gespeichert. Im Rahmen des Build-Prozesses (Compilieren) wird aus ihr durch den Qt eigenen Präprozessor die entsprechende C++ konforme Header-Datei erstellt, die durch Einbinden in den Quelltext nach Programmstart alle entsprechenden Objekte des Hauptfensters erstellt, so wie es die Abbildung 4.2 zeigt.

Den größten Bereich des Hauptfensters nimmt dabei die „Konsole“, ein Objekt vom Typ `QTextEdit`, ein. Durch ihr erfolgen sämtliche Ausgaben des Hauptprogramms und der aufgerufenen Funktionen. Daten und berechnete Werte werden dabei in grüner, Fehlermeldungen und sonstige Programmmeldungen in roter Schriftfarbe dargestellt, um sie so besser zu unterscheiden. Überschreiten die Ausgaben in der Konsole die Objekthöhe, so wird automatisch ein Scrollbalken eingeblendet. Mit Scrollen lassen sich so auch vorangegangene Ausgaben nochmals einsehen.

Die einzelnen Schritte der verteilten Speicherung mittels Erasure Resilient Coding, gemäß dem Prinzip in Abb. 2.3, werden dabei durch fünf Funktionen simuliert. Aufgerufen wird jede einzelne Funktion durch Betätigung einer der Schaltflächen am linken Rand des Hauptfensters. Mittels der Qt-eigenen „signals and slots“-Technik wird dann in das entsprechende Unterprogramm verzweigt und einer der folgenden Prozesse abgearbeitet.

Select File: Mittels einer Dialog-Box wird ein beliebiges „source file“ ausgewählt. Der Inhalt der Datei wird als Text in der Konsole ausgegeben.

Split: Die zuvor ausgewählte Datei wird in m gleichgroße Stücke geteilt. Die Fragmente werden nun in m neue Dateien gespeichert. Der Inhalt der einzelnen Fragmente wird über die Konsole spaltenweise ausgegeben.

Encode: Hier erfolgt die Berechnung der k redundanten Datenblöcke durch Anwendung des Reed-Solomon-Encodings. Diese werden anschließend in k neue Dateien gespeichert.

Loss: Hier werden einige der in den letzten beiden Stufen entstandenen neuen Dateien gelöscht.

Decode: Im letzten Schritt werden alle zuvor neu erstellten und nicht gelöschten Dateien in den Arbeitsspeicher geladen. Die fehlenden Dateien werden durch das Reed-Solomon-Decoding rekonstruiert. Alle ursprünglichen m Dateiblöcke werden dann in der richtigen Reihenfolge in einer neuen Datei geschrieben.

Diese Unterprogramme haben dabei als Gemeinsamkeit lediglich Zugriff auf eine einzige globale Variable, die als eigene Klasse instantiiert ist und aus den folgenden vier Komponenten besteht:

- `int mchunks`; Anzahl der m -Chunks
- `int nchunks`; Anzahl der n -Chunks
- `int state`; Status
- `QString sourcefile`; Dateiname des „source files“ inkl. Pfad

Die Variable `state` dient dabei der Implementierung einer finite state machine, mit deren Hilfe zum einen der Fortschritt des Erasure Resilient Coding optisch in Form der fünf Checkboxes durch Setzen der Häkchen nach erfolgreicher Abarbeitung angezeigt wird. Zum anderen ist sie Grundlage für eine eingebaute Sicherheitssperre. So ist es beispielsweise nicht möglich, die Datei zu Teilen noch bevor überhaupt Eine ausgewählt wurde. Entsprechende Fehlgriffe werden über die Konsole mit Fehlermeldungen quittiert.

Die grundlegenden Parameter des Reed-Solomon-Codes lassen sich über die Schaltfläche „Config“ wählen. Im sich daraufhin öffnenden Konfigurations-Dialog bleibt die Wahl der Parameter auf die Anzahl der ursprünglichen m -Chunks und den redundanten k -Chunks begrenzt. Die daraus resultierende Anzahl der n -Chunks und der Faktor S werden bei Änderung der Grundparameter berechnet und im Konfigurations-Dialog angezeigt. Bricht man die Einstellung der Parameter mit Betätigung der Schaltfläche „Cancel“ ab, so behalten die zuvor eingestellten Parameter ihre Gültigkeit. Bei Bestätigung der eingestellten Werte mittels der Schaltfläche „Apply“, werden diese als neue Parameter in der globalen Variable übernommen und in die Datei „config.cfg“ geschrieben. Diese wird nach einem Programmstart innerhalb der Initialisierungsphase geöffnet und ausgelesen, um die zuletzt gesetzten Parameter weiterhin zu nutzen. Kann diese Datei nicht für den Lesezugriff geöffnet werden, so werden die Default-Werte mit $m = 4$, $k = 2$ gesetzt.

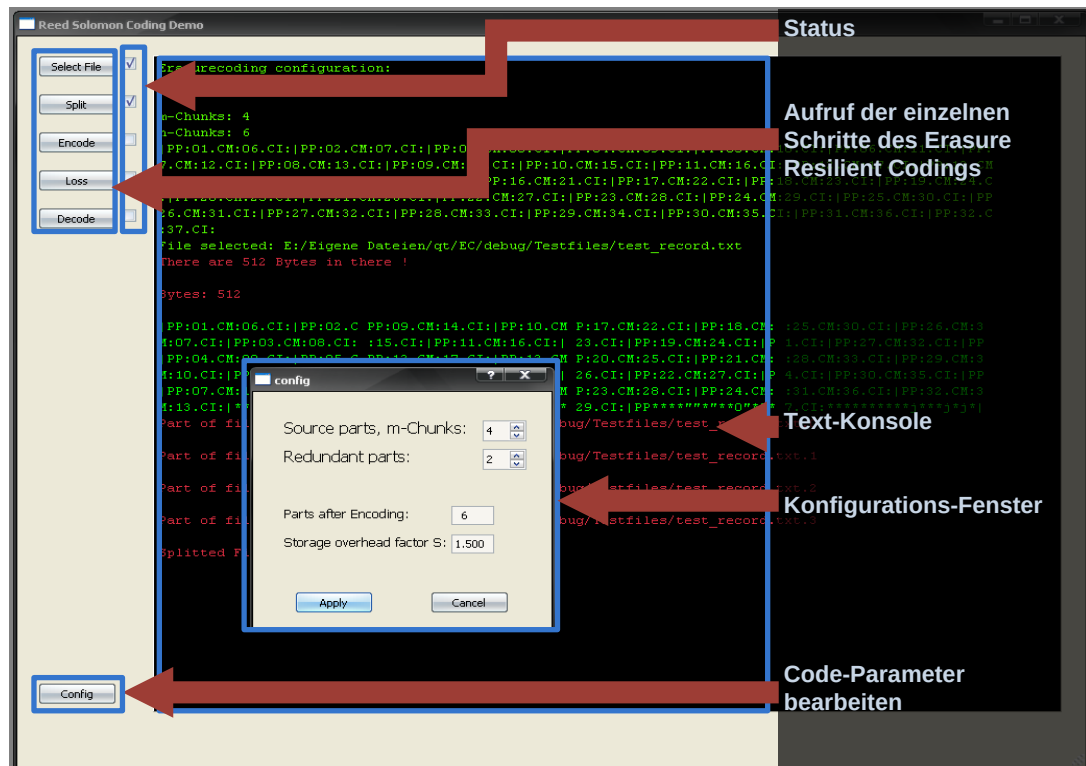


Abbildung 4.2: Bedienelemente und Funktionen des Hauptfensters

4.3 Unterprogramm: Select File

Unmittelbar nach Betätigen der Schaltfläche „Select File“ öffnet sich ein Dialogfenster, mit dessen Hilfe sich eine beliebige Datei auswählen lässt. Wird die Dateiauswahl bestätigt, so wird der Name der Datei inklusive vollem Pfad in der globalen Variablen `sourcefile` gespeichert. Damit ist allen Unterprogrammen (Methoden) der Hauptanwendung der Dateiname und der Speicherort bekannt, so dass alle Prozesse des Erasure Resilient Coding auf ihr angewandt werden können.

Nach erfolgter Auswahl der Datei, wird diese umgehend geöffnet und ihr Inhalt Byte-weise ausgelesen und auf der Konsole dargestellt. Während des Auslesens der Datei werden die Bytes mitgezählt und am Ende der Ausgabe mitangegeben. Da bei entsprechend großen Dateien die Ausgabe des Inhalts auf der Konsole zu langen Verzögerungszeiten führt, wird die Ausgabe nach 5000 Bytes abgebrochen.

4.4 Unterprogramm: Split

In diesem Unterprogramm erfolgt das Zerteilen der zuvor ausgewählten Datei in gleichgroße Datenblöcke. Dazu wird im ersten Schritt die Dateigröße und damit die Anzahl der in ihr enthaltenen Zeichen mittels des Qt-eigenen Befehls `bytesAvailable()`; ermittelt. Die Größe eines Datenblocks ergibt sich dann aus:

$$\text{splitlen} = \frac{\text{Dateigröße}}{\text{Anzahl der m-Chunks}} \quad (4.1)$$

Dabei gilt zu beachten, dass hier mit Ganzzahlwerten (`integer`) gerechnet wird, d.h., eventuell vorhandene Nachkommastellen fließen nicht in die Berechnung der Datenblockgröße mit ein. Es muss also auch der Rest r der Division (Dateigröße modulo Anzahl der m-Chunks) betrachtet werden. Ist der Rest $r = 0$, so ist der aktuelle Wert von `splitlen` die Größe der zu erstellenden Datenblöcke. Ging die Division nicht vollständig auf, d.h., Rest $r > 0$, so wird das Divisionsergebnis `splitlen` um Eins erhöht. Dadurch soll verhindert werden, dass die letzten r Zeichen der Ausgangsdatei bei der Dateiteilung verloren gehen. Im nächsten

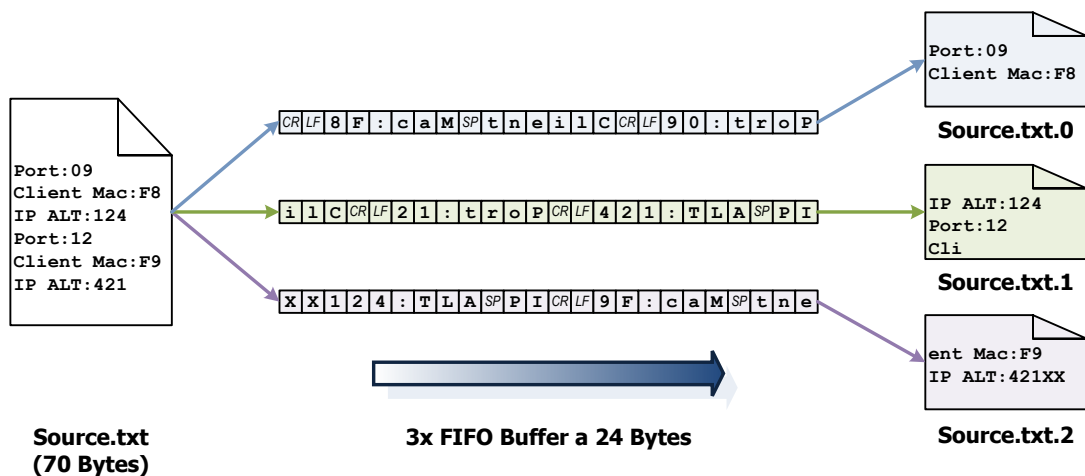


Abbildung 4.3: Grundprinzip der Dateiteilung bei $mchunks = 3$

Schritt werden Vektoren in der Anzahl der globalen Variablen `mchunks` und der Länge `splitlen` angelegt. Im Laufe der Implentierung stellte sich heraus, dass der Gebrauch von großen Arrays zu Problemen hinsichtlich ihrer Dimensionierung führt, da das Anlegen derer auf dem Stack geschieht, der aber nur wenige MBytes umfasst. Um dieser Speicherbegrenzung entgegen zu wirken, erfolgt die Definition daher als Vektoren, hier realisiert anhand der Qt Klassenbibliothek `QVector`. Diese werden auf dem Heap angelegt und sind in ihrer Dimension nur durch die Leistungsfähigkeit des jeweils verwendeten Betriebssystems begrenzt. Diese dienen nun der Aufnahme der Zeichen der Ausgangsdatei und stehen für die zukünftigen

Datenblöcke. Ihre Funktionsweise entspricht einem Puffer mit der Funktion des First In – First Out (FIFO). In der Implementierung sind sie sinnvollerweise zu einem zweidimensionalen Vector mit den Dimensionen $[x][y]=[\text{splitlen}][\text{mchunks}]$ zusammengefasst.

Die Ausgangsdatei wird nun für den Lesezugriff geöffnet und Byte-weise ausgelesen. Dabei werden aufeinanderfolgende Bytes in der Anzahl `splitlen` in ein und demselben Puffer gespeichert. Nach dem Auslesen der Datei erfolgt noch eine Behandlung der Reststellen des letzten Puffers, die nicht Bestandteil der Ausgangsdatei sind und mit zufälligen Werten belegt sind, sofern $r > 0$ ist. Dabei werden die $\text{mchunks} - r$ letzten Stellen des letzten Puffers mit dem `integer` Wert 88, welches dem ASCII-Zeichen „X“ entspricht, überschrieben und maskiert.

Die so erzeugten `mchunks` Datenblöcke werden nun in der Konsole spaltenweise ausgegeben. Um auch hier lange Wartezeiten durch die Ausgabe großer Datenmengen zu vermeiden, beschränkt sich die Ausgabe auf maximal 30 Zeilen. Auf die Ausgabe wird ganz verzichtet, wenn die Anzahl der `mchunks` sehr hoch gewählt wird, da die Spaltenbreite dann zu gering ausfällt. Ebenso werden sämtliche Werte, die Steuer- und Sonderzeichen darstellen, herausgefiltert und gegen „*“ ersetzt.

Im letzten Schritt wird jeder Puffer als eigenständige Datei gespeichert. Dazu wird der Puffer als eindimensionaler Vector an eine weitere Methode der Hauptklasse, namentlich `writefile`, übergeben, die das Speichern der Datei übernimmt. Der neue Dateiname wird dabei auf Grundlage des in der globalen Variablen `sourcefile` hinterlegten Namens, durch Anfügen der Puffernummer als weitere Dateiextension erzeugt. Damit sind `mchunks` neue Dateien entstanden, deren Dateieindungen von 0 bis $\text{mchunks} - 1$ durchnummeriert sind.

Abbildung 4.3 zeigt exemplarisch den gesamten Vorgang der Dateiteilung.

4.5 Unterprogramm: Encode

4.5.1 Erstellung des Erweiterungskörpers

Grundlage für die Anwendung des Reed-Solomon-Codes ist der Erweiterungskörper $GF(2^{k_1})$, in welchem alle Berechnungen stattfinden werden. Ausgangspunkt für seine Erstellung ist ein geeignetes Modularpolynom. Da es sich in dem hier vorliegendem Fall um Bytes handelt, die codiert werden sollen, ist $k_1 = 8$ zu wählen, da $1\text{Byte} = 8\text{Bit}$ sind, und somit steht der *grad* des Modularpolynoms mit

$\text{grad}(M(x)) = 8$ fest. Damit lassen sich maximal 255 unterschiedliche Symbole codieren. Um die maximale Symbolanzahl nutzen zu können, muss also auch der Zyklus der Code-Wiederholung maximiert werden, womit am Modularpolynom auch der Anspruch der Primitivität besteht. Das Modularpolynom $M(X)$ wird daher der Tabelle 3.2 entnommen und lautet:

$$M(X) = x^8 + x^6 + x^5 + x^4 + 1 \quad (4.2)$$

Dieses wird in einen eindimensionalen Array der Dimension $x = 9$ gespeichert, indem jede vorhandene Stelle des Polynoms eine binäre 1 repräsentiert. Das Array enthält also die binäre Folge: 101110001.

Nun werden zwei weitere Arrays deklariert: $\text{GF}[8][256]$ dient zum Speichern der binären Koeffizienten der Restpolynome, $\text{GFwerte}[256]$ dient der Aufnahme der Dezimalwerte, die durch die binären Koeffizienten entstehen. Der laufende Index $[0...255]$ der beiden Arrays steht dabei für die zu codierenden Symbole und lässt sich auch als Potenz von α ausdrücken durch $\text{Index} = \alpha^{\text{Index}-1}$. Die ersten neun

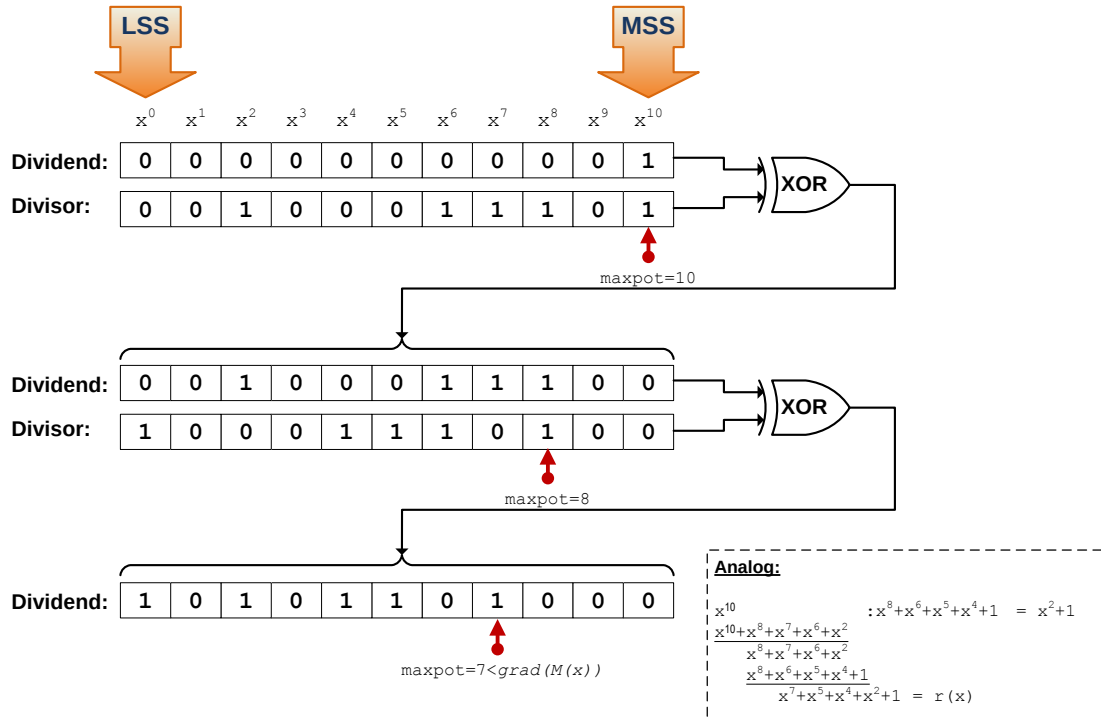


Abbildung 4.4: Berechnung des Restpolynoms am Beispiel von α^{10}

Werte des Erweiterungskörpers lassen sich einfach direkt setzen, da eine Polynomdivision erst ab $\text{Index} = 9 = \alpha^8 = x^8$ möglich ist. $\text{Index} = 0$ bleibt dabei das Nullelement mit binärer 0, $\text{Index} = 1$ bis $\text{Index} = 8$ entstehen durch Zeilenweises verschieben der binären 1. Alle weiteren Elemente werden mittels binärer

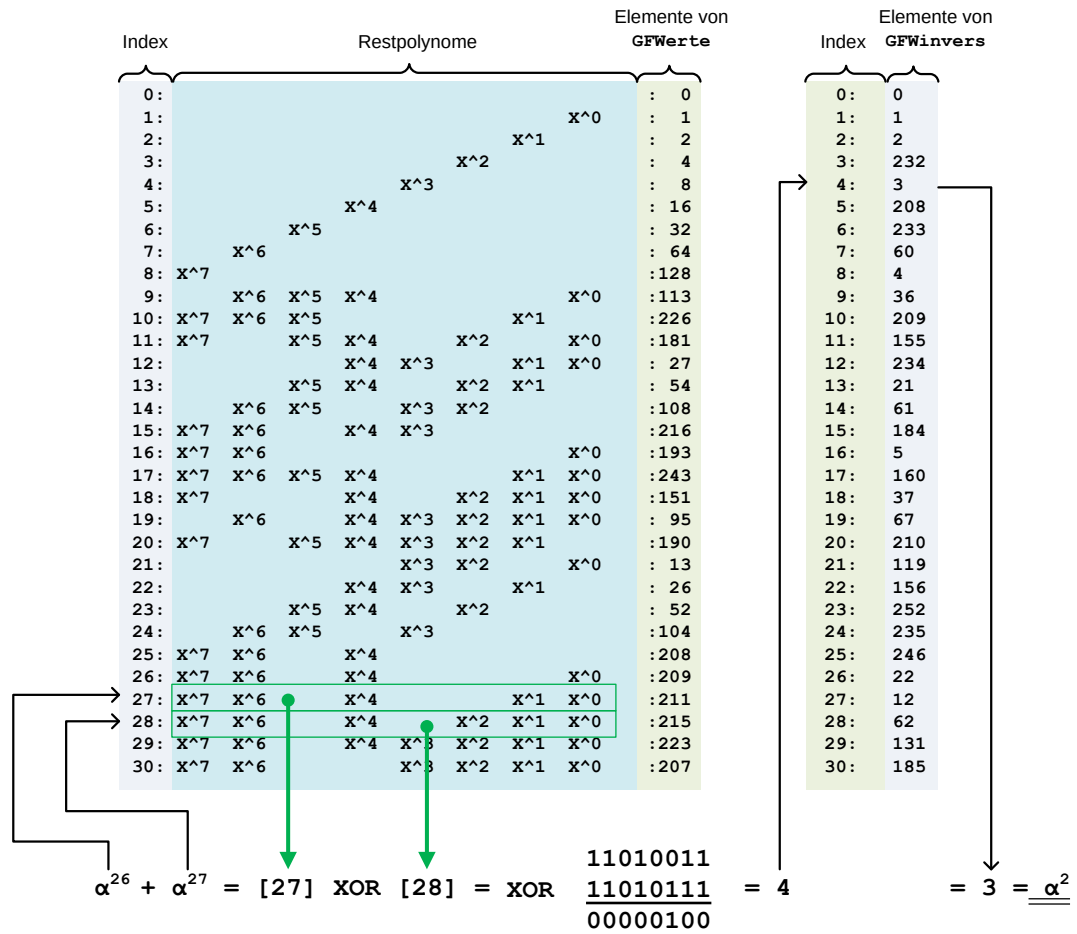


Abbildung 4.5: Grund- und Erweiterungskörper über $GF(2^8)$

Polynomdivision (innherhalb von $GF(2)$) berechnet. Dazu wird in einem weiteren eindimensionalen Array der Dimension 256, welches als Dividend dient, eine binäre 1 an der Position der Wertigkeit des zu behandelnden Symbols gesetzt. Diese Position wird auch als Ausgangspunkt für den Divisionszeiger **maxpot** festgelegt. Ein weiteres Array gleicher Dimension, der Divisor, wird ab der Position **maxpot** mit den binären Koeffizienten des Modularpolynoms befüllt. Somit stehen Dividend und Divisor an der Stelle der höchsten auftretenden Potenz **maxpot** direkt untereinander. Dies ist praktikabel, da an dem eigentlichem Ergebnis der Polynomdivision kein Interesse besteht. Dividend und Divisor werden nun von hinten beginnend, elementweise mit der XOR-Funktion verknüpft, welches der Subtraktion entspricht, und das Ergebnis wird umgehend im Dividenten gespeichert. Der Divisionszeiger **maxpot** zeigt dabei auf die letzte Position, bei der die XOR-Verknüpfung eine binäre 1 ergab. Damit ist auch die Position der höchsten auftretenden Potenz bekannt. Der Divisor wird wiederum mit dem Modularpoly-

nom an der Position `maxpot` befüllt und es erfolgt die nächste XOR-Verknüpfung. Dieser Vorgang wiederholt sich solange, bis der Wert `maxpot` den *grad* des Modularpolynoms unterschreitet: $\text{maxpot} < \text{grad}(M(x))$. Eine weitere Division ist dann nicht mehr möglich. Das Array des Dividenden enthält dann die Koeffizienten des Restpolynoms, die dann in das Array `GF` eingetragen werden. Abbildung 4.4 zeigt anhand eines Beispiels die durchgeführten Operationen. Sind die Restpolynome für alle Symbole berechnet, wird der dezimale Wert anhand der Summenbildung der binären Einsen entsprechend ihrer Wertigkeit 2^{Position} berechnet und in das Array `GFwerte` eingetragen. Damit ist der Erweiterungskörper zu $GF(2^8)$ erstellt.

Da in den noch folgenden Berechnungen jede Addition/Subtraktion von Symbolen mittels einer XOR-Verknüpfung durchgeführt wird und das Ergebnis ein Element des Erweiterungskörpers ist, nicht aber das Symbol selbst, wird ein weiteres Array `GFwinvers` mit gleicher Dimension des Arrays `GFwerte` angelegt. Dieses hat als Index die Werte des Erweiterungskörpers und enthält die Werte, die den Index des Arrays `GFwerte` darstellen. Dies erspart die Suche innerhalb des Erweiterungskörpers nach dem äquivalenten Symbol. Ein Ausschnitt dieser beiden Arrays, inklusive eines kurzen Beispiels zur Anwendung der beiden Arrays anhand eines Rechenbeispiels mit XOR-Operation, soll in Abbildung 4.5 gezeigt werden.

Bei den Abbildungen von Arrays und den in ihnen enthaltenen Werten zeichnet sich aber auch gleichfalls ein Problem ab: Ohne Angabe der Indices, ist es schwer, die Wertigkeit der einzelnen Symbole zu bestimmen. Dies ist aber von grundlegender Bedeutung, da die Symbole Koeffizienten innerhalb eines Polynoms mit entsprechender Position in Form einer Potenz von x darstellen. Ein Vertauschen der Reihenfolge der Koeffizienten führt unweigerlich zu Fehlinterpretationen. Deshalb werden im folgenden zwei Begriffe, in Anlehnung an Registerbeschreibungen, eingeführt, die die Wertigkeit der Symbole innerhalb des Arrays definieren:

MSS: Most Significant Symbol, Position der höchstwertigsten Potenz

LSS: Least Significant Symbol, Position der niederwertigsten Potenz

4.5.2 Berechnung des Generatorpolynoms

Von grundlegender Bedeutung bei allen zyklischen Codes ist das Generatorpolynom, mit dessen Hilfe die Codierung durchgeführt wird. Der *grad* dieses Polynoms entspricht dabei der Anzahl der redundanten Symbole: $\text{grad}(g(x)) = k$. Somit

muss $k > 0$ sein, ansonsten wird kein Generatorpolynom erstellt, und es wird keine Codierung durchgeführt. Mit der Bildungsformel 3.11 ist das Generatorpo-

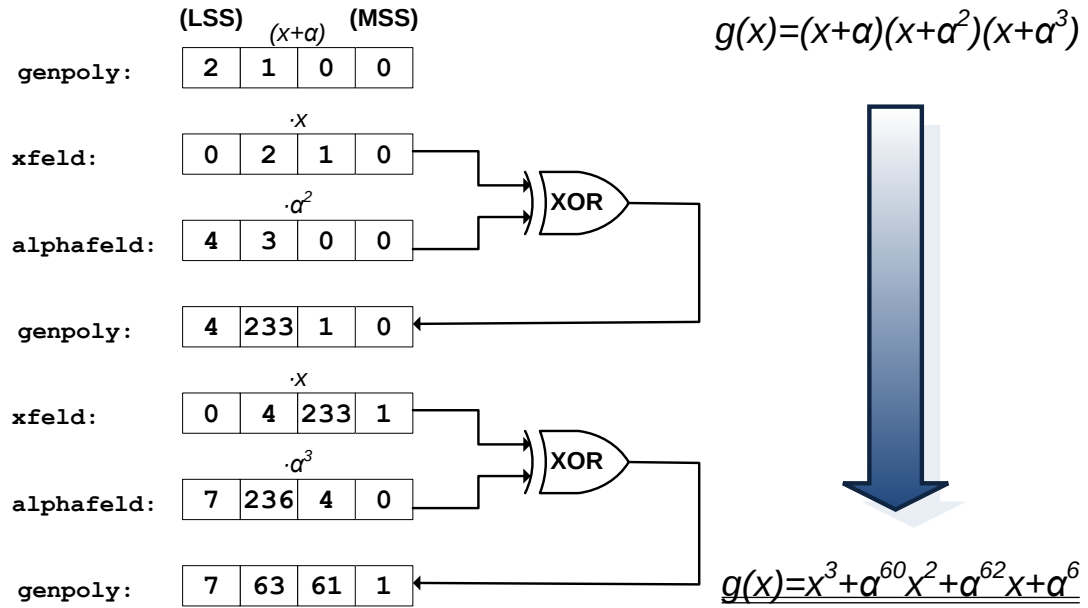


Abbildung 4.6: Algorithmus zur Erzeugung des Generatorpolynoms

lynom als Produkt von Linearfaktoren bereits gegeben. Doch für den Vorgang der Codierung muss das Generatorpolynom in Polynomschreibweise vorliegen, es muss also ausmultipliziert werden. Dazu wird im ersten Schritt das eindimensionale Array **genpoly** mit der Dimension $k + 1$ angelegt, das die Zwischenergebnisse der Multiplikation und das Endergebnis enthält. Die erste Nullstelle des Generatorpolynoms $(x + \alpha)$ wird dabei direkt eingetragen. Wurde die Anzahl der redundanten Chunks mit $k = 1$ gewählt, so ist das Generatorpolynom bereits vollständig und es finden keine weiteren Berechnungen mehr statt. Bei $k > 1$ werden noch zwei weitere Arrays, **xfeld** und **alphafeld**, angelegt, die als Hilfsfelder für Zwischenberechnungen während einer Multiplikation dienen. Bei der Multiplikation mit der nächsten Nullstelle von $g(x)$ werden dazu im ersten Schritt alle Elemente des Arrays **genpoly** in das Array **xfeld** kopiert. Die Multiplikation mit dem x eines Nullstellenpolynoms bedeutet dabei lediglich eine Verschiebung aller Elemente des Arrays **xfeld** um eine Stelle in Richtung des MSS. Anschließend werden alle Elemente der vorangegangenen Multiplikation, das sind die Elemente des Arrays **genpoly**, mit dem α^i des aktuellen Nullstellenpolynoms multipliziert. Dabei handelt es sich um eine Addition der Exponenten von α und kann vollständig im Grundkörper $GF(2^8)$ durchgeführt werden. Dazu werden beide Symbole um Eins dekrementiert, man erhält die entsprechenden Exponenten von α , und es erfolgt die Addition der beiden Exponenten. Das Ergebnis wird im Anschluss um Eins inkrementiert und im Hilfsfeld **alphafeld** abgespeichert. Ei-

der jeweiligen FIFO-Puffer zusammengesetzt und direkt in das Array des Dividenden, in eine um den $\text{grad}(g(x))$ verschobenen Position in Richtung des MSS, eingetragen. Das Generatorpolynom wird nun direkt an der Position der höchsten auftretenden Potenz des Dividenden als Divisor eingetragen. Die Polynomdivision erfolgt nun genauso, wie sie bereits zur Berechnung der Restpolynome des Erweiterungskörpers $GF(2^8)$ durchgeführt wurde. Nur erfolgt sie jetzt nicht mit binären Koeffizienten, sondern mit Elementen des $GF(2^8)$. Dabei werden die bereits besprochenen Rechenregeln innerhalb des Erweiterungskörpers angewandt. Auch hier wird die Division beendet, sobald der Divisionszeiger `maxpot` eine Position in Richtung des LSS erreicht, die kleiner als der grad des Divisors $\text{grad}(g(x))$ ist. Die k Koeffizienten des so entstandenen Restpolynomes bilden dabei die redundanten Symbole und werden symbolweise in richtiger Reihenfolge auf die dafür vorgesehenen FIFO-Puffer verteilt.

Abschließend werden die FIFO-Puffer, welche die redundanten Symbole enthalten, in Form eindimensionaler Vektoren an die Methode `writefile` übergeben. Diese übernimmt das Abspeichern eines jeden Puffers in einer extra Datei. Damit entstehen genau k neue Dateien. Die Dateinamen werden dabei weiterhin nummeriert, beginnend mit der nächsten Nummer des letzten Datenblocks.

Das Grundschema der Codierung kann der Abbildung 4.7 entnommen werden.

4.6 Unterprogramm: Loss

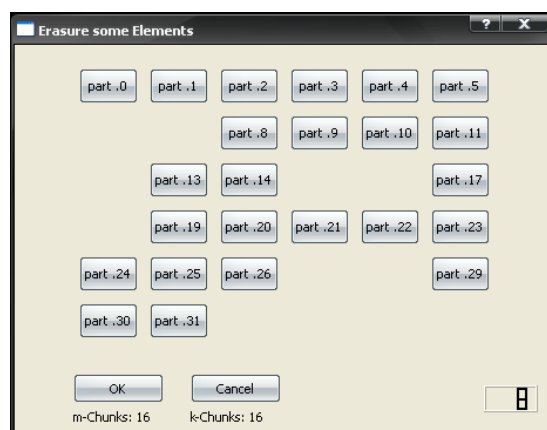


Abbildung 4.8: Dialog zum Löschen einiger Dateiblöcke

Die Aufgabe dieses Unterprogramms ist es, mittels eines Dialoges ausgewählte Dateiblöcke zu löschen. Dieser Vorgang simuliert das Auslöschen einiger Chunks, bzw. stellt deren Nichtverfügbarkeit beim Versuch dar, auf die auf andere Teil-

nehmer des P2P-Netzwerkes verteilten Chunks zuzugreifen.

Dazu öffnet sich ein separates Dialogfenster, in dem alle bis dahin erstellten n Chunks in Form von Buttons dargestellt werden, wie es in Abbildung 4.8 zu sehen ist. In der unteren rechten Ecke des Dialoges befindet sich eine LCD-Anzeige, ein Qt Widget von Typ `QLCDNumber`, die die aktuelle Anzahl der zum Löschen vorgemerkten Dateiblöcke anzeigt. Wird nun einer der Buttons betätigt, der den entsprechenden Dateiblock symbolisiert, wird die Methode `ack` ausgeführt. Dabei wird die laufende Nummer des Buttons mittels eines gemappten Signals an die Methode übergeben. Diese dient der aufgerufenen Methode zur Identifikation des betätigten Buttons und ermöglicht es, den entsprechenden Dateiblock zur Löschung vorzumerken. Dies geschieht durch Eintrag eines `True` in dem boolschen Löschvektor `delfield`, einem eindimensionalen Array der Klasse `lost`. Weiterhin wird der Zählerstand der LCD-Anzeige um eins erhöht. Dabei wird auch geprüft, ob die Anzahl der zum Löschen vorgemerkten Dateien p die Anzahl der redundanten Dateiblöcke k überschreitet. Im Falle von $p > k$ kann eine vollständige Rekonstruktion der Ausgangsdatei ausgeschlossen werden, welches nun durch eine in rot gehaltene LCD-Anzeige signalisiert wird. Der betätigte Button wird nun noch mittels `findChild` als Objekt identifiziert und gelöscht. Er wird nun dauerhaft aus dem Dialogfenster entfernt.

Mit Betätigung des Buttons `OK` erfolgt die Ausführung der Methode `lostapply`. Alle nun im Löschvektor maskierten Dateiblöcke werden mittels `remove()` dauerhaft vom Datenträger entfernt, und das Dialogfenster schließt sich. Wird hingegen der Button `Cancel` betätigt, wird die bis dahin erstellte Auswahl verworfen und das Dialogfenster umgehend geschlossen. Es findet kein Löschen von Dateien statt.

4.7 Unterprogramm: Decode

Der letzte Programmteil hat die Aufgabe, die im vorangegangenen Schritt ausgelöschten Dateiblöcke wiederherzustellen. Da die dazu notwendigen Berechnungen wiederum im Erweiterungskörper $GF(2^8)$ stattfinden, werden, so wie es in Abschnitt 4.5.1 bereits beschrieben wurde, wiederum die Arrays `GFwerte` und `GFwinvers` angelegt und deren Inhalt berechnet.

Nun wird die Existenz der zuvor n erstellten Datenblöcke überprüft. Dies geschieht auf Grundlage der Information, die in der globalen Variablen `sourcefile` gespeichert ist, somit sind Speicherort und Name der Ausgangsdatei bekannt. Die

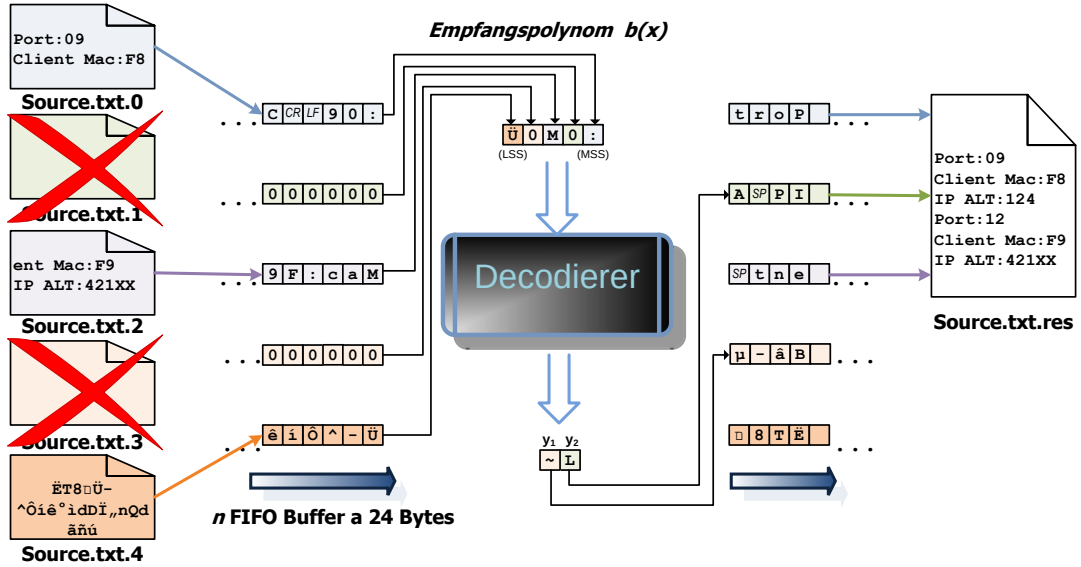
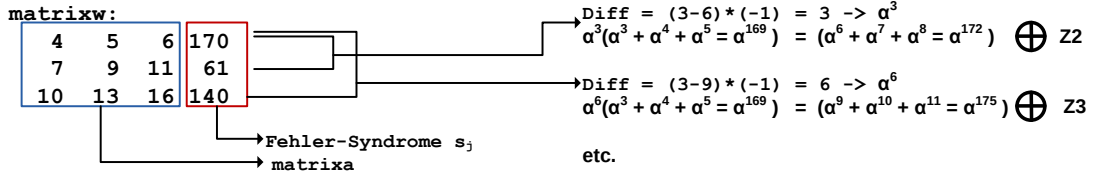


Abbildung 4.9: Schemata der Decodierung bei: $m = 3, k = 2, n = 5$

Namen der Dateiblöcke ergeben sich dann aus dem Namen der Ausgangsdatei plus dem durchnummerierten Suffix von 0 bis $n - 1$ als Dateiendung. Die Qt-Funktion `file.exists()`; überprüft dabei das Vorhandensein einer Datei und liefert ein `false`, im Falle des Fehlens, und ein `true` für die Existenz der abgefragten Datei. Kann die Datei nicht gefunden werden, so wird im Auslöschvektor `miss`, ein eindimensionales Array der Dimension n , der fehlende Datenblock an entsprechender Position, durch Eintragen der entsprechenden Potenz von α , maskiert. Weiterhin wird die Variable `missings` um eins erhöht, sie dient dem Zählen der insgesamt fehlenden Chunks. Gefundene Dateien werden dabei mit einer 0 in entsprechender Position im Auslöschvektor `miss` eingetragen. Der erste gefundene Dateiblock wird dabei auch gleich auf seine Größe hin untersucht und als Dimension zum Anlegen der n FIFO-Puffer genutzt. Die vorhandenen Dateiblöcke werden nun in den Speicher geladen, wobei jeder genau einen FIFO-Puffer füllt. Die Puffer der fehlenden Dateiblöcke werden dabei auf NULL gesetzt. In der Implementierung werden die FIFO-Puffer wiederum als zweidimensionaler Vector mit den Dimensionen für x =Dateiblockgröße und $y = n$ realisiert.

Nun werden die Elemente des Auslöschvektors, welche die fehlenden Dateiblöcke symbolisieren, in einem weiteren Array `xfeld`, in der Dimension der Anzahl der fehlenden Blöcke, eingetragen. Für das im Verlauf der Decodierung noch aufzustellende lineare Gleichungssystem, wird das Array `matrixa` in den Dimensionen $x = y$ =Anzahl der fehlenden Datenblöcke angelegt und deren Werte gemäß der Formel 3.16 berechnet. Dieser Vorgang der Decodierung wird an dieser Stelle vorgezogen, da alle Parameter für die Berechnungen von x_j^i bereits bekannt sind und

Ausgangssituation: Lineares Gleichungssystem



Nach der Eliminationsphase: Rücksubstitution

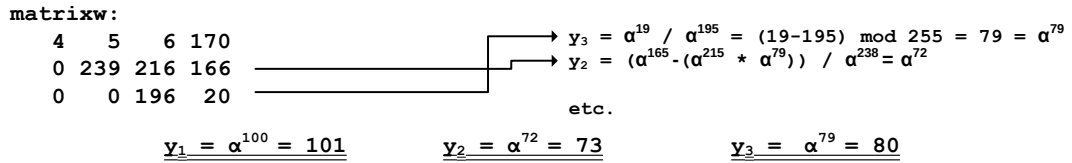


Abbildung 4.10: Gauß-Algorithmus im Falle des Fehlens von drei Chunks

diese sich auch im Laufe der Decodierung nicht mehr ändern.

Die nun folgenden Schritte dienen der Wiederherstellung der ausgelöschten Symbole und müssen daher in der Anzahl der Symbole pro FIFO-Puffer durchgeführt werden.

Im ersten Schritt wird das Empfangspolynom $b(x)$ erstellt. Dabei wird jeweils ein Symbol den FIFO-Puffern entnommen und der Wertigkeit nach in das Array **bx** eingetragen. An den Stellen der Auslöschungen steht dabei automatisch eine 0. Nun erfolgt die Berechnung der Fehlersyndrome s_j entsprechend der Anzahl der fehlenden Datenblöcke p gemäß der Bildungsformel 3.15. Dazu werden die Elemente des Empfangspolynoms $b(x)$ in das Hilfsarray **bxw** kopiert. Nun wird jedes Symbol mit dem Symbol $\alpha^{j*Stellung}$ multipliziert, was der Addition der Potenzen von α entspricht. Nun werden alle Elemente des Hilfsfeldes addiert. Dies geschieht mittels XOR-Verknüpfung innerhalb des Erweiterungskörpers $GF(2^8)$. Das resultierende Ergebnis wird als Fehlersyndrom s_j im Array **syndrom** gespeichert. Sind alle Syndrome berechnet, sind auch alle Parameter zur Aufstellung des Gleichungssystems vorhanden.

Zur Berechnung der Symbole der Auslöschstellen y_j muss nun noch ein lineares Gleichungssystem gelöst werden. Dies geschieht in der Implementierung mit dem gaußschen Eliminationsverfahren. Dazu wird die quadratische Matrix **matrixa** um eine Spalte ergänzt, die die Syndrome enthält. Das nun vollständige Gleichungssystem ist dabei in dem Array **matrixw** gespeichert. Im ersten Schritt deren Lösung steht die Vorwärtselemination, wodurch das Gleichungssystem eine Stufenform erhält. Dazu wird im ersten Schritt die Differenz der Exponenten von α des ersten Symbols der ersten Zeile und dem darunterliegendem Symbol der

gleichen Spalte berechnet und das Vorzeichen umgekehrt. Diese Differenz wird auf alle Exponenten der Symbole der ersten Zeile aufaddiert und die gesamte Zeile mit ihren Elementen mit denen der zweiten Zeile XOR-verknüpft und das Resultat dort eingetragen. Danach ist das erste Element der zweiten Zeile eine 0 und damit ausgelöscht. Das Gleiche geschieht dann mit der ersten und dritten Zeile usw., bis alle Elemente der ersten Spalte ab der zweiten Zeile ausgelöscht sind. Dies wird nun Zeilen- und Spaltenweise so weitergeführt, bis die gewünschte Stufenform des Gleichungssystems erreicht ist. Nun erfolgt die Rücksubstitution. Der erste gesuchte Wert y kann direkt berechnet werden durch die Subtraktion der Exponenten des letzten x -Elementes der letzten Zeile von dem letzten Syndrom, welches das Element der letzten Spalte und Zeile ist. Dies entspricht der Division der beiden. Das Resultat wird im Ergebnisarray `yfeld` gespeichert und zur weiteren Berechnung der darüberliegenden Zeile genutzt. Die Ergebnisse aus dem Array `yfeld` werden dabei mit den entsprechenden x -Elementen des Gleichungssystems multipliziert (Addition ihrer Exponenten), vom Syndrom der jeweiligen Zeile subtrahiert (XOR-Verknüpfung) und anschließend durch das erste Symbol der Zeile dividiert (Subtraktion ihrer Exponenten), bis alle Werte y_j berechnet sind. Ein Fallbeispiel ist dazu in Abbildung 4.10 zu sehen. Das Empfangspolynom $b(x)$ wird nun mit den fehlenden Symbolen an den ausgelöschten Stellen ergänzt. Nun werden alle Symbole wieder auf die FIFO-Puffer entsprechend ihrer Position verteilt und die ursprünglichen Daten sind vollständig hergestellt.

Die ersten m Puffer werden der Reihe nach in eine Datei geschrieben, deren Dateiendung `.res` (restored) lautet. Sofern kein Decodierungsversagen auftrat, ist die ursprüngliche Ausgangsdatei vollständig und fehlerfrei wiederhergestellt. Das Grundschaema der Decodierung kann der Abbildung 4.9 entnommen werden.

4.8 Verifikation

Die Funktion der Implementierung wurde hauptsächlich durch funktionales Testen nachgewiesen. Dabei wurden verschiedenste Dateitypen in unterschiedlicher Größe auf verschiedenen Computersystemen durch die Implementierung codiert und nach Löschen einiger Dateiblöcke wieder decodiert, mit positivem Ergebnis. Die Codeparameter m und k wurden dabei ständig variiert.

Der Test bezog sich also nicht auf Textdateien allein, da sich in ihnen meist nur ein eingeschränkter Symbolvorrat aus dem Alphabet A befindet. Um eine ordnungsgemäße Codierung/Decodierung nachzuweisen, wurden auch binäre Da-



Abbildung 4.11: Beispiel einer binären Codierung mit RS(32,16)-Code

teien, wie etwa Bitmap-Grafiken und ausführbare Dateien, dem Test unterzogen. Diese Datei-Typen reagieren sehr empfindlich auf kleinste Bit-Fehler. So zeugen Pixelfehler in einer rekonstruierten Bitmap-Datei schon rein optisch von Decodierungsversagen. Veränderungen in ausführbaren Dateien führen meist zum Absturz derer. Mit Hilfe der Abbildung 4.11 lässt sich auch sehr gut verdeutlichen, wie das Resultat des Decodierens im Falle von mehr als k Auslösungen ausfällt. Nur die nicht von der Auslöschung betroffenen Dateiteile der m -Chunks zeigen nach dem Decodieren noch Teile der Grafik, die rekonstruierten Teile zeigen nur noch Rauschen als Zeichen des vollständigen Decodierungsversagens.

Um jedoch sicher zu gehen, dass es zu keinen Veränderungen zwischen Ausgangs- und decodierter Datei kam, wurden zwei Softwaretools benutzt, die den Inhalt zweier Dateien miteinander vergleichen und Fehler maskieren. Zum einen steht unter dem Betriebssystem **MS Windows XP** das Tool **fc** zur Verfügung, deren Aufruf in der Konsole erfolgt. Schneller und komfortabler bietet aber auch der Dateimanager **Total Commander** die Möglichkeit, den Inhalt zweier Dateien zu vergleichen.

Da die Möglichkeiten eines funktionalen Tests beschränkt sind, wurden nicht alle Kombinationen aus m, k , Dateityp und -größe getestet.

4.9 Datendurchsatz

Im Folgenden wurde die Implementierung auf verschiedenen PC-Systemen getestet und die Zeiten, die für Codierung und Decodierung benötigt werden, gemessen.

Auf Grundlage dieser und den entsprechenden Dateigrößen, kann so die mittlere Datenrate in MBit/s berechnet werden. Folgende PC-Systeme, auf allen kommt das Betriebssystem **MS Windows XP** zum Einsatz, wurden dabei verwendet:

PC1: Intel Core 2 Duo T7700 @ 2,4 GHz, 2 GB Ram

PC2: AMD Athlon64 4000+ @ 2,5 GHz, 2 GB Ram

PC3: Intel Pentium III @ 750 MHz, 256 MB Ram

Für diese und alle weiteren Tests, bei denen die Codierungs-/Decodierungszeiten bestimmt wurden, wurde der Quelltext dahingehend modifiziert, dass während der Codierung/Decodierung keine Ausgaben auf der Konsole erfolgen, da sie die Zeitmessungen beeinflussen. Die Zeitnahme erfolgte jeweils mit Hilfe der Qt-Klassenbibliothek **QTime**. Dabei wurden fünf unterschiedlich große Dateien, 4 Textdateien und eine binäre Bitmap-Datei, mittels RS(32,16)-Code codiert, und nach Löschen der ersten 8 Chunks erfolgte die Decodierung. Die dabei erzielten Messwerte können den Tabellen 4.1 und 4.2 entnommen werden.

Dateigröße	PC1		PC2		PC3	
	Zeit /ms	MBit/s	Zeit /ms	MBit/s	Zeit /ms	MBit/s
128 KB	125	8,4	187	5,6	460	2,3
256 KB	250	8,4	375	5,6	931	2,3
512 KB	516	8,1	750	5,6	1853	2,3
1024 KB	1031	8,1	1469	5,7	3745	2,2
2,04 MB	2109	8,1	3000	5,7	7711	2,2
		Ø=8,2		Ø=5,6		Ø=2,3

Tabelle 4.1: Datendurchsatz des Encoding-Algorithmus

Dateigröße	PC1		PC2		PC3	
	Zeit /ms	MBit/s	Zeit /ms	MBit/s	Zeit /ms	MBit/s
128 KB	62	16,9	78	13,4	251	4,2
256 KB	110	19,1	156	13,4	491	4,3
512 KB	235	17,8	297	14,1	1002	4,2
1024 KB	469	17,9	609	13,8	2023	4,1
2,04 MB	937	18,3	1234	13,9	4146	4,1
		Ø=18		Ø=13,7		Ø=4,2

Tabelle 4.2: Datendurchsatz des Decoding-Algorithmus

Wie nicht anders zu erwarten, ist die Datenrate dabei nicht von der Größe der jeweiligen Datei und auch nicht vom Dateityp abhängig. Da durch die Implementierung der Algorithmen eine Vielzahl von Rechenoperationen durchgeführt werden, hängt die erzielte Datenrate ausschließlich von der zur Verfügung stehenden Rechenleistung des jeweiligen Systems ab.

4.10 Komplexität Encoding und Decoding

Um eine Aussage über die Komplexität des Codier- und des Decodiervorgangs treffen zu können, wurde der Test mit der 1024 KByte großen Textdatei fortgesetzt. Hierbei wurde die Anzahl der m -Chunks mit $m = 16$ konstant gehalten, während die Anzahl der redundanten Chunks k im Intervall $[2..30]$ variiert wurde. Nach jedem Codiervorgang wurden dann genau k der ersten Chunks gelöscht und die Decodierung durchgeführt. Das Ergebnis der Messreihe kann als grafische Darstellung der Abbildung 4.12 entnommen werden.

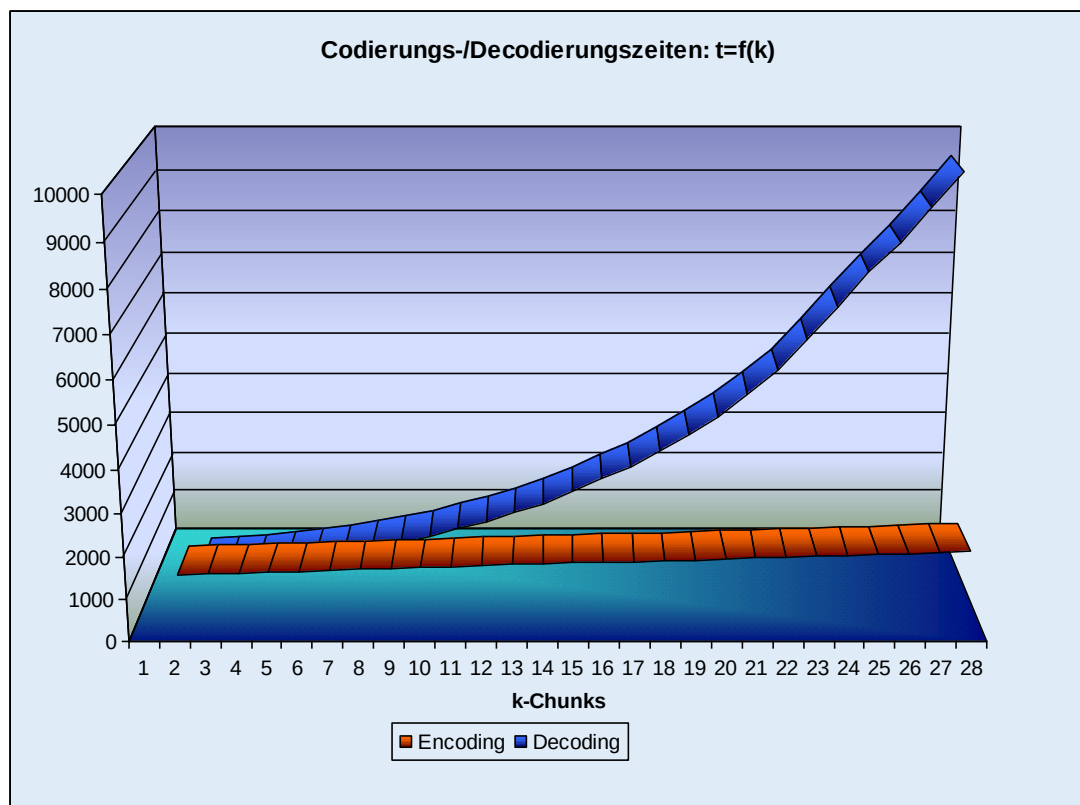


Abbildung 4.12: Codierungs-/Decodierungszeiten am Beispiel eines $RS(16+x,16)$ -Codes

Dabei zeigt der Codiervorgang ein rein lineares Verhalten bei Erhöhung der k -Chunks. Ursache dafür ist, dass zur Polynomdivision lediglich der Divisor um jeweils k Symbole erweitert wird, die Anzahl der Divisionsschritte bleibt dabei konstant. Die Zeit, die zum Decodieren benötigt wird, steigt dagegen quadratisch mit der Anzahl der Auslöschstellen $p = k$ an. Hauptursache dafür ist das Lösen des linearen Gleichungssystems zur Berechnung der ausgelöschten Symbole. Mit jeder Erhöhung der Auslöschstelle p um Eins, vergrößert sich die Matrix des Gleichungssystem um eine Spalte und eine Zeile, auf die dann die Lösungsstufen der Eliminationsphase und der Rücksubstitution angewandt werden müssen.

5 Zusammenfassung und Ausblick

Durch den Einsatz der verteilten Speicherung im P2P-Netzwerk lässt sich die Verfügbarkeit von Daten deutlich steigern. Aber nur durch den Einsatz der Erasure Resilient Codes lassen sich die Kosten, die durch das Hinzufügen von Redundanz zur ursprünglichen Datenmenge entstehen, in einem verträglichen Rahmen halten.

Dabei sticht eine Codierungsform mit seinen herausragenden Eigenschaften und seiner Flexibilität ganz besonders hervor: der Reed-Solomon-Code. Mit seiner Entdeckung vor fast 50 Jahren, gehört er heute zu einem der am besten erforschten Codes und ist aus dem heutigen Alltag nicht mehr wegzudenken. Neben seiner „perfekt“ fehlerkorrigierenden Fähigkeit, überzeugt er vor allem auch durch seine Eigenschaft der „optimalen“ Rekonstruktion von Auslöschstellen. Dabei finden die Reed-Solomon-Codes bereits jetzt eine weite Verbreitung als Erasure Resilient Code in RAID-Level-6 Festplattenverbundsystemen und werden auch in dem zur Zeit im Prototyp-Stadium befindlichen globalen Speichersystem OceanStore [8] eingesetzt werden.

Lange Zeit sprach der immense „Hunger“ an Rechenleistung dieser Codierungsform, hervorgerufen durch die komplexen Berechnungen innerhalb eines abgeschlossenen Erweiterungskörpers, einer weiten Verbreitung als Erasure Resilient Code entgegen. Seit 1995, seit Beginn der eigentlichen Definition der Erasure Resilient Codes, ist man in verschiedenen Bereichen auf der Suche nach Alternativen, die eine beinahe gleiche Optimalität hinsichtlich aufgewendeter Redundanz zur Wiederherstellungsfähigkeit bei gleichzeitiger Verringerung der Komplexität gewährleisten. Vor allem die Firma Digital Fountain mit ihrer Fountain-Code-Familie und IBM mit den Weaver-Code [1][3] haben auf diesem Gebiet gravierende Fortschritte gemacht.

Aber schon bei der hier in Software vorliegenden Implementierung des Reed-Solomon-Codes wird deutlich, dass akzeptable Ergebnisse möglich sind. Dabei handelt es sich im Grunde genommen um eine Umsetzung der Algorithmen aus dem Lehrbuch, fast ohne jede Optimierung. Aber selbst daraus wird ersichtlich,

dass durch die jeweiligen Rechnersysteme zur Verfügung gestellte Rechenleistung, die von Jahr zu Jahr immer noch steigt, ausreichend ist, um zumindest kleine bis mittlere Datenmengen in angemessener Zeit durch verkürzte RS-Codes zu codieren und zu decodieren. Eine weitere Option wäre die Implementierung der Algorithmen in Hardware. Seit einigen Jahren bietet sich die Programmierung der Field Programmable Gate Arrays (FPGA) in einer hardwarebeschreibenden Sprache (HDL) aufgrund ständig sinkender Preise geradezu an, um auf eine flexible Art in zumindest geringen bis mittleren Stückzahlen diese rechenintensive Algorithmen zu implementieren und somit ein Höchstmaß an Datendurchsatz zu erzielen.

Literaturverzeichnis

- [1] Kannan Ramchandran Alexandros G. Dimakis, Vinod Prabhakaran. *Decentralized Erasure Codes for Distributed Networked Storage*. University of California, 2004.
- [2] Peter Herbig Bernd Friedrichs. *Kanalcodierung: Grundlagen und Anwendungen in modernen Kommunikationssystemen*. Springer-Verlag Berlin Heidelberg, 1996.
- [3] James Lee Hafner. *WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems*. IBM Almaden Research Center, 2005.
- [4] Markus Hufschmid. *Information und Kommunikation Grundlagen und Verfahren der Informationsübertragung*. Teubner, 2007.
- [5] Nokia Corporation in Finland. <http://qt.nokia.com/products>, 2009.
- [6] Mark Summerfield Jasmin Blanchette. *C++ Programming With Qt4*. Prentice Hall, 2006.
- [7] Thomas Heinen Klaus Manhart. *Achtung, IT-Disaster*. <http://www.itmittelstand.de>, 2009.
- [8] Johann Latocha. *OceanStore*. Albert-Ludwigs-Universität Freiburg, 2007.
- [9] Andre Neubauer. *Kanalcodierung*. J. Schlemmbach Fachverlag, 2006.

6 Erklärung über die selbständige Anfertigung der Arbeit

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte Hilfe Dritter verfasst sowie keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Angaben, die inhaltlich oder wörtlich aus fremden Werken stammen, wurden kenntlich gemacht. Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde, Fachhoch- oder Hochschule vor und wurde ebenfalls bisher noch nicht veröffentlicht.

Rostock, 15. Oktober 2009

Maik Gotzmann