Bachelorarbeit

Komprimierungs- und Aggregationsverfahren für Nachrichten in drahtlosen Sensornetzwerken

Themenersteller: Prof. Dr.-Ing. Jörg Nolte
Betreuer: Dipl. Inf. Reinhardt Karnapke
vorgelegt von: Steffen Schulze Matrikel Nr. 2105182
Abgabetermin: 16.10.2006
Inhaltsverzeichnis

Abbildungsverzeichnis......................................................................................................5
1 Einleitung.............................................................................................................................9
  1.1 Zielsetzung der Arbeit.................................................................................................10
  1.2 Gliederung der Arbeit...............................................................................................10
2 Analyse.............................................................................................................................13
  2.1 Statistik einer Nachrichtenquelle...............................................................................13
  2.2 Grundlagen zur Datenkomprimierung......................................................................15
  2.3 Arten der Redundanz..................................................................................................16
    2.3.1 Lauflängenkodierung.........................................................................................18
    2.3.2 Wortkodierung.....................................................................................................22
    2.3.2.1 LZ77...................................................................................................................23
    2.3.2.2 LZSS..................................................................................................................26
    2.3.2.3 LZ78..................................................................................................................27
    2.3.2.4 LZW..................................................................................................................30
    2.3.3 Entropiekodierung...............................................................................................32
    2.3.3.1 Shannon-Fano-Kodierung...............................................................................33
    2.3.3.2 Huffman-Kodierung.......................................................................................35
    2.3.3.3 Arithmetische Kodierung................................................................................39
    2.3.4 Unterstützende Verfahren...................................................................................42
    2.3.4.1 Burrows-Wheeler-Transformation.................................................................42
    2.3.4.2 Move-To-Front-Transformation.......................................................................46
  2.4 Kombination der Komprimierungsalgorithmen............................................................47
  2.5 Eignung der Algorithmen entsprechend der Anforderungen der Arbeit....................49
  2.6 Verfahren zur Aggregation........................................................................................51
3 Entwurf einer Aggregationsschicht für COPRA............................................................53
  3.1 Rahmenwerk der Arbeit.............................................................................................53
    3.1.1 REFLEX...............................................................................................................53
    3.1.2 COPRA...............................................................................................................53
    3.1.3 SERNET...............................................................................................................53
  3.2 Planung der Umsetzung...............................................................................................54
  3.3 Vorhalten sendebereiter Pakete..................................................................................56
  3.4 Maße der Praxistauglichkeit......................................................................................56
  3.5 Integration ins Rahmenwerk......................................................................................57
  3.6 Interner Aufbau der Compression-Schicht.................................................................57
4 Implementierung.............................................................................................................59
  4.1 Implementierung der Komprimierungsalgorithmen......................................................59
  4.2 Einbindung der Komprimierungsalgorithmen in die Netzwerkschicht.......................62
4.3 Implementierung der Aggregation ........................................62
4.4 Zusammenspiel von Aggregation und Komprimierung .........63
4.5 Optimierungen in der Implementierung ..............................63
4.6 Beispielkonfiguration der Compression-Schicht ...............63

5 Bewertung ...........................................................................67
  5.1 Aufbau und Ablauf der Simulationen in SERNET..............67
    5.1.1 Betrachtung der Paketanzahl pro Anwendungsdurchlauf......70
    5.1.2 Betrachtung des gesendeten Datenvolumens ..................73
    5.1.3 Detaillierte theoretische Betrachtung ausgesuchter Experimente......74
    5.1.4 Testszenario mit 25 Knoten .........................................76
    5.1.5 Messtabellen ................................................................78
  5.2 Aufbau und Ablauf der Experimente auf dem RCX ............79
    5.2.1 Betrachtung der Kodegröße .........................................79
    5.2.2 Betrachtung der Laufzeit und des Energieverbrauchs ..........80
  5.3 „Burst“-Verhalten der Aggregation ....................................84
  5.4 Fazit der Bewertungen .....................................................86

6 Ausblick ..............................................................................87
Abbildung 2.48.: Vergleichsmaße der Kodierung aus dem Huffman-Algorithmus. 38
Abbildung 2.49.: Übermittelte Daten des Kodierungsbaums aus dem
Huffman-Beispiel. 39
Abbildung 2.50.: schematische Teilintervalle der Arithmetischen Kodierung. 40
Abbildung 2.51.: numerische Teilintervalle der Arithmetischen Kodierung. 40
Abbildung 2.52.: Eingangsdaten für Burrows-Wheeler-Transformation. 42
Abbildung 2.53.: Liste aller Rotationen der Eingangsdaten vor der Sortierung. 43
Abbildung 2.54.: Liste aller Rotationen der Eingangsdaten nach der Sortierung. 43
Abbildung 2.55.: Ausgangsdaten für Burrows-Wheeler-Transformation. 43
Abbildung 2.56.: Rücktransformation nach Burrows-Wheeler im ersten Schritt. 44
Abbildung 2.57.: Rücktransformation nach Burrows-Wheeler im zweiten Schritt. 44
Abbildung 2.58.: Rücktransformation nach Burrows-Wheeler im dritten Schritt. 45
Abbildung 2.59.: Rücktransformation nach Burrows-Wheeler im letzten Schritt. 45
Abbildung 2.60.: Beispiel einer Move-To-Front-Transformation. 47
Abbildung 2.61.: Vergleich der Entropie vor und nach der
Move-To-Front-Transformation. 48
Abbildung 2.62.: Redundanz- und ausgaben der Komprimierungsalgorithmen. 49
Abbildung 2.63.: Redundanz- und ausgaben der Komprimierungsalgorithmen. 50
Abbildung 2.64.: Rechenaufwand und Zusatzinformationsanteil der
Komprimierungsalgorithmen. 51
Abbildung 3.1.: Nachrichtendaten mit verschiedenen Alphabeten betrachtet. 56
Abbildung 3.2.: Möglicher Aufbau einer PPE mit Compression-Schicht. 57
Abbildung 3.3.: Basisaufbau der Compression-Schicht. 57
Abbildung 4.1.: Klassendeklaration des Interface CompressCodec. 59
Abbildung 4.2.: Klassendeklaration von RleCodec. 60
Abbildung 4.3.: Klassendeklaration von LzwCodec. 60
Abbildung 4.4.: Klassendiagramm HuffmanCodec. 61
Abbildung 4.5.: Einbindung der Compression-Schicht in eine PPE. 64
Abbildung 4.6.: Basiskonfiguration in der CompressConfig.h. 64
Abbildung 4.7.: Detailkonfiguration in der CompressConfig.h. 65
Abbildung 4.8.: Komprimierungskonfiguration in der CompressConfig.h. 66
Abbildung 5.1.: Knotenanordnung in 6nodes. 69
Abbildung 5.2.: Knotenanordnung in line6. 69
Abbildung 5.3.: Referenzmessung für line6. 70
Abbildung 5.4.: Paketanzahl für Testfall line6. 71
Abbildung 5.5.: Referenzmessung für 6nodes. 72
Abbildung 5.6.: Paketanzahl für Testfall 6nodes. 72
Abbildung 5.7.: Datenvolumen für Testfall line6. 73
Abbildung 5.8.: Datenvolumen für Testfall 6nodes. 74
Abbildung 5.9.: Formeln der real übertragenen Präambeldaten. 75
Abbildung 5.10.: Formeln des real übertragenen Datenvolumens. 75
Abbildung 5.11.: Abschätzung der möglichen Byte-Operationen pro Aggregation. 75
Abbildung 5.12.: Knotenanordnung in web25. 76
Abbildung 5.13.: Paketanzahl für Testfall web25. 77
Abbildung 5.14.: Datenvolumen für Testfall web25. 78
Abbildung 5.15.: Komplette Messdaten für 6nodes. 78
Abbildung 5.16.: Komplette Messdaten für line6. 78
Abbildung 5.17.: Komplette Messdaten für web25. 79
Abbildung 5.18.: Messdaten der Kodegröße auf dem RCX. 80
Abbildung 5.19.: Messdaten der Laufzeit auf dem RCX in Ticks. 81
Abbildung 5.20.: Relation eines Ticks zu Prozessorzyklen. 82
Abbildung 5.21.: Anhand der Aggregation gesparte Byte je Sendevorgang. 82
Abbildung 5.22.: Durch Komprimierung und Aggregation gesparte Byte je Sendevorgang........................................................................................................................................82
Abbildung 5.23.: Formel der Arbeitsleistung des Controllers in Abhängigkeit zu Ticks..................................................................................................................................................82
Abbildung 5.24.: Formel der Arbeitsleistung des Funkmoduls in Abhängigkeit zu den gesendeten Byte...........................................................................................................................................83
Abbildung 5.25.: Zusätzliche und gesparte Arbeitsleistungen des Testaufbaus.............................83
Abbildung 5.26.: Durchschnittliche Energieeinsparung pro Einzelpaket......................................84
Abbildung 5.27.: Einfaches „Burst“-Verhalten der Aggregation....................................................85
Abbildung 5.28.: „Burst“-Verhalten auf einem Teilkad.................................................................85
1 Einleitung


1.1 Zielsetzung der Arbeit

Diese Arbeit befasst sich mit Techniken der Aggregation und Komprimierung in tief eingebetteten Systemen und wird die Charakteristika dieser für den Einsatz im Netzwerkprotokollstack von COPRA (siehe Kapitel 3) überprüfen. Ziel ist die praktische Umsetzung als eine anpassungsfähige und transparente Schicht für den Netzwerkprotokollstack, welche die Nachrichten durch Aggregation und Komprimierung aufbereitet. Dies verspricht einen geringeren Energieverbrauch des Funkverkehrs, ohne dabei dessen Funktionalität negativ zu beeinflussen.

1.2 Gliederung der Arbeit

Die Arbeit ist in 5 weitere Kapitel aufgeteilt. In Kapitel 2 wird ein Überblick gegeben, welche Ansätze es in der Komprimierung und Aggregation gibt. Hier wird bereits kurz auf die Einsatzmöglichkeiten der Algorithmen für die gestellte Aufgabe der Bachelorarbeit eingegangen. Im dritten Kapitel wird ein Entwurf für die Implementierung erstellt. Es werden Entscheidungen getroffen, welche Algorithmen umgesetzt werden können und in welchem Rahmenwerk dies geschieht.
Das vierte Kapitel beschreibt dann die endgültige Implementierung und welche Änderungen möglicherweise vorgenommen werden müssen. Um das Erreichte einschätzen zu können, findet im fünften Kapitel eine Bewertung nach objektiven Kriterien wie Größe der Umsetzung und Einfluss auf den Netzverkehr statt. Den Abschluss findet die Arbeit in einer Zusammenfassung möglicher Probleme und Erfolge sowie einem Ausblick auf zukünftige Möglichkeiten.
2 Analyse

In diesem Kapitel werden Komprimierungs- und Aggregationsverfahren betrachtet, um später eine objektive Auswahl zu treffen, welche Algorithmen im Rahmen dieser Bachelorarbeit umgesetzt werden.

2.1 Statistik einer Nachrichtenquelle


Eine Quelle hat eine bestimmte Anzahl $n$ von verschiedenen Symbolen, welche als Symbolvorrat bezeichnet wird (Abbildung 2.1).

![Abbildung 2.1: Symbolvorrat](image)

$m : Nachrichtenmenge pro Symbol
n = 2^m$

$H_0 := m = \log(n)$

$H_0$ ist der Entscheidungsgehalt der Quelle oder auch die mittlere minimale Symbollänge $m$ pro Symbol/Zeichen (Einheit bit/Symbol), die sich aus der Anzahl der verschiedenen Symbole ergibt (Abbildung 2.2).

![Abbildung 2.2: Formel des Entscheidungsgehalts $H_0$](image)

Die einzelnen Symbole/Zeichen einer Quelle haben eine bestimmte Auftrittswahrscheinlichkeit $p_i$ zwischen 0 und 1. Bei einer Wahrscheinlichkeit $p_i=0$ tritt ein Zeichen $i$ nie auf, bei $p_i = 1$ immer. Die Summe aller Wahrscheinlichkeiten ist 1 (Abbildung 2.3).
Der Begriff der Entropie wurde für die Informationstheorie durch Claude Elwood Shannon etwa zur Hälfte des letzten Jahrhunderts eingeführt [2]. Sie ist für eine Nachricht das Maß des Mittleren Informationsgehalts. Nimmt sie den Wert 1 an, kommen alle Zeichen gleich häufig vor. Die Entropie $H$ berechnet sich wie folgend in Abbildung 2.4 zu sehen ist.

Die Entropie $H$ kann höchstens so hoch wie der Entscheidungsgehalt $H_0$ werden. Daher wird im Zusammenhang mit dem Entscheidungsgehalt in einigen Quellen auch von der Maximalentropie gesprochen (Abbildung 2.5).

Als Redundanz $R$ bezeichnet man die Differenz zwischen dem Entscheidungsgehalt und der Entropie einer Nachrichtenquelle. Da der Entscheidungsgehalt nicht kleiner als die Entropie sein kann, ist die Redundanz nie negativ. Sind $H_0$ und $H$ gleich groß, enthält die Nachricht keine Redundanzen (Abbildung 2.6).

Die mittlere Kodewortlänge $H_c$ einer Nachricht berechnet sich ähnlich wie die Entropie,
nur dass man für jedes Symbol die Bitlänge der Symbolkodierung direkt einsetzt. \( H_c \) ist die entscheidende Größe für die Einschätzung von Entropiekodierungen (Abbildung 2.7).

\[
H_c = -\sum_{i=1}^{n} p_i \cdot m_i
\]

Abbildung 2.7.: Formel der mittleren Kodewortlänge

2.2 Grundlagen zur Datenkomprimierung

Als Datenkomprimierung, oder auch Datenkompression, bezeichnet man Verfahren zur Reduktion des Speicherbedarfs von gegebenen Daten. Zu solchen Verfahren gehört die Komprimierung, welche die Daten zusammenpackt, und die Dekomprimierung, die die kodierten Daten wieder in den Ausgangszustand überträgt.

Man unterscheidet grundsätzlich zwischen allgemeiner verlustfreier und anwendungsspezifischer verlustbehafteter Komprimierung. Im verlustfreien Fall können die Ausgangsdaten ohne Verlust des Informationsgehalts wiederhergestellt werden. Dabei werden bei der Komprimierung lediglich Redundanzen beseitigt. Solche Komprimierungen sind unabhängig von der Anwendung, da sie keine besonderen Informationen über die Nutzung der Daten erfordern.

Dem gegenüber stehen verlustbehaftete Verfahren, in denen der Informationsgehalt der Ausgangsdaten aus der komprimierten Darstellungsform nicht mehr komplett wiederhergestellt werden kann. Daten, die auf der Empfängerseite unwichtig sind, werden nicht mitkodiert. In solchen Fällen spricht man von einer Irrelevanzreduktion. Dies ist der Fall bei einigen Bildkomprimierungsalgorithmen wie zum Beispiel JPEG oder auch bei dem weit verbreiteten MP3-Audiocodec. In beiden Fällen werden Daten, die die menschlichen Sinne nicht wahrnehmen, bei der Komprimierung schlichtweg entfernt.

Da bei der Datenübertragung im Fall des Netzwerkprotokollstacks kein Informationsverlust erwünscht ist, sind für die aktuelle Betrachtung nur verlustfreie Verfahren relevant.

Nicht alle Komprimierungsalgorithmen erlauben das strombasierte Bearbeiten von Daten. Die Nachricht wird dabei nicht in einem Paket komprimiert, sondern die
Komprimierung beginnt bevor alle Nachrichtendaten eingetroffen sind. Entropiekodieren müssen zu Beginn die Verteilungshäufigkeiten bekannt sein oder diese werden erst im Laufe der Verarbeitung dynamisch ermittelt, um ein Zeichen anhand seiner Kodewortlänge zu codieren. Der Protokollstack überreicht die Daten allerdings in Paketform, wodurch dieser Aspekt außer Acht gelassen werden kann. Im folgenden werden bei Zeichen immer die Werte von 0 bis 255 oder äquivalent dazu die ASCII Zeichen angenommen. Dies entspricht der Symbolmenge eines Byte und damit der kleinsten direkt adressierbaren Einheit in verbreiteten Computerumgebungen.

2.3 Arten der Redundanz

Man kann Redundanzen grob in folgende Gruppen untergliedern:

**Wiederholung von Einzelsymbolen** ist die vielleicht am einfachsten zu erkennende Redundanz, auch für den menschlichen Blick. Ein Beispiel ist in der Abbildung 2.8 dargestellt.

| Beispiel: AAAAAAAABBBBBCCCCDDDDAAAABBBBCCCCCDDDDAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
Die **Symbolverteilung** kann zum Komprimieren genutzt werden, wenn zum Beispiel abhängig von der Häufigkeit den Symbolen unterschiedlich lange Repräsentationen gegeben wird. In der Abbildung 2.11 tritt das Symbol A häufiger auf als B und C.

<table>
<thead>
<tr>
<th>Beispiel: ABACABACABACABACABAC</th>
</tr>
</thead>
</table>

*Abbildung 2.11.: Beispiel Symbolverteilung*
2.3.1 Lauflängenkodierung

Bei der Lauflängenkodierung werden Symbolwiederholungen in eine kürzere Darstellungsform überführt, vergleichbar mit der Multiplikation. Im Englischen nennt sich das Verfahren "RunLength Encoding", woher die im Folgenden genutzte Abkürzung RLE stammt.

Bei der Lauflängenkodierung werden Symbolwiederholungen in eine kürzere Darstellungsform überführt, vergleichbar mit der Multiplikation. Im Englischen nennt sich das Verfahren "RunLength Encoding", woher die im Folgenden genutzte Abkürzung RLE stammt.

In Abbildung 2.12 ist bereits auf den ersten Blick eine Wiederholung einzelner Zeichen zu erkennen. Eine Schreibweise wie in Abbildung 2.13 bietet sich hierbei an.

Das Multiplikationszeichen kann als offensichtlich weggelassen werden, wodurch sich die Zeichenfolge, wie in Abbildung 2.14 ergibt.


Für diese Testfolge bedeutet dies eine weitere Verbesserung der Komprimierung um ein Zeichen weniger auf 13.


Wie festgestellt, lohnt sich eine Kodierung von Einzelzeichen nicht. Zusätzlich kann auch die Kodierung von 2er oder 3er Folgen gespart werden. Bei Ersteren benötigt es mehr, bei den Zweiten genauso viel Daten. Dadurch brauchen bei den Kodierungszeichen für die Anzahl die Fälle 0, 1, 2 und 3 nicht beachtet zu werden. Dies gibt die Möglichkeit einiger Optimierungsoptionen.

Die erste wäre ein Zeichen, das dem Escape-Zeichen entspricht, mit dem Zweiertupel „Escape-Zeichen – 0“ zu kodieren, was den Nachteil der Sonderdarstellung von 3 auf 2 verringert. Allerdings schon ab 2 konsekutiven Zeichen, die dem Escape-Zeichen entsprechen, lohnt sich wieder das Dreiertupel.


Ein derartiger RLE-Algorithmus kann seine Vorteile bei Daten mit sich häufig wiederholenden Zeichen ausspielen. Dies ist der Fall bei einfachen Rasterbildern mit wenig Farbabstufungen und klaren Kanten.

Bei Daten mit wenig Wiederholungen ist er überfordert und führt im ungünstigsten Fall durch die höhere Darstellungslänge bei kurzen Escape-Zeichenfolgen zu einem größeren Speicherbedarf.
Detailoptimierungen lässt der Algorithmus auch noch auf Bitebene zu. Allerdings wären diese im Kontext auf Effektivität und Machbarkeit zu überprüfen.


Nicht selten nutzt eine Nachricht nicht die volle Menge der zur Verfügung stehenden Symbole von 256, sondern beispielsweise nur zwei Drittel und diese ohne Ordnung. In solchen Fällen kann über eine feste Abbildung nachgedacht werden, um die Zeichen auf den Wertebereich von 0 aufwärts zu ordnen. Der obere Symbolbereich (255 abwärts) wird dadurch weniger genutzt und führt daher zu weniger Sonderfällen bei der Kodierung. Damit wären die Daten ideal für den vorangegangenen Optimierungsgedanken der Lauflängenkodierung.

Fazit:
Daher empfiehlt sich die Lauflängenkodierung als Vorstufe vor einer weiteren Komprimierung, um die Eingabedaten vorzubereiten.
Das Verfahren ist frei von Patentansprüchen.
2.3.2 Wortkodierung


Liegt zum Beispiel als zu komprimierende Eingabe der Text in der Abbildung 2.16 vor, so wird folgendes Verzeichnis angelegt (Abbildung 2.17):

```
begin
print Hallo Welt
print Hallo World
end
```

*Abbildung 2.16.: Eingangsnachricht für Wortkodierung*

1: begin
2: print
3: Hallo
4: Welt
5: World
6: end

*Abbildung 2.17.: Wörterbuch für Wortkodierung*

Mit dessen Hilfe kann nun der Text neu kodiert werden, wie er in Abbildung 2.18 dargestellt ist.

```
1
2 3 4
2 3 5
6
```

*Abbildung 2.18.: Kodierte Nachricht der Wortkodierung*

Wie man im Beispiel erkennen kann, wurde der Eingangstext gut zusammengekürzt. Dabei muss allerdings beachtet werden, dass das Verzeichnis dem Dekodierer bekannt sein muss. Dazu ist es nötig, das Verzeichnis mit zu übertragen, solange beide Seiten sich nicht ein festes Verzeichnis teilen.

2.3.2.1 LZ77


Der Algorithmus sucht nun die in der Vorschau betrachtete Symbolfolge im Suchfenster. Findet er diese oder einen Teil derer beginnend beim ersten Zeichen, gibt er ein Tripel mit der Startposition (P) im Suchfenster, der Zeichenlänge der Phrase (L) und dem Folgesymbol (S) der Phrase in der Vorschau aus. Wird die Phrase im Verzeichnis nicht gefunden, schreibt er nur das Tripel 0,0,<neues Symbol> in die Ausgabe. Dieses Vorgehen kann noch optimiert werden, indem man die Vorschau für die Phrasenfindung mit einbezieht. Näheres dazu im Beispiel unten.

Im Folgenden wird dies mit der Eingangsnachricht “ANANAS” demonstriert. Das Suchfenster hat eine Größe von 8, das Vorschaufenster von 4 Symbolen.
Zu Beginn (Abbildung 2.19) ist das Suchfenster leer, da noch keine Symbole kodiert wurden. In der Vorschau werden die ersten 4 Symbole betrachtet. Aufgrund des leeren Suchfensters wird das erste Symbol als Einzelzeichen ausgegeben mit dem Tripel 0,0,A. Es wurde damit 1 Symbol kodiert. Um genau diese Schrittzahl wird auch das betrachtete Fenster weitergeschoben.

<table>
<thead>
<tr>
<th>Suchfenster</th>
<th>Vorschau</th>
<th>Ausgabe</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 P L S</td>
<td>A N A N 0 0 A</td>
</tr>
</tbody>
</table>

*Abbildung 2.19.: Kodierungsschritt 1 LZ77*

In Abbildung 2.20 befindet sich im Suchfenster das erste Symbol der Eingangsnachricht, das A. Die Vorschau betrachtet die Phrase „NANA“, welche auch noch nicht im Verzeichnis gefunden werden kann. Damit wird das N als einzelnes Symbol kodiert mit dem Tripel 0,0,N. Es wurde 1 Symbol kodiert, womit auch das Fenster um 1 Schritt weiter wandert.

<table>
<thead>
<tr>
<th>Suchfenster</th>
<th>Vorschau</th>
<th>Ausgabe</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 P L S</td>
<td>A N A N A 0 0 N</td>
</tr>
</tbody>
</table>

*Abbildung 2.20.: Kodierungsschritt 2 LZ77*

damit ein Tripel gespart.

Die Ausgabe sieht dann wie folgt aus (Abbildung 2.22):

\[
\begin{array}{cccccccc}
0 & 0 & A & 0 & 0 & N & 6 & 3 & S \\
\end{array}
\]

\textit{Abbildung 2.22.: Kodierte Nachricht des LZ77 Algorithmus}

Das Dekodieren ist eine ressourcensparende Angelegenheit. Die Tripel der Kodierung können bei Einzelsymbolen direkt in die Ausgabe geschrieben werden, bei Phrasen handelt es sich lediglich um eine Kopieraktion von bereits dekodierten Zeichenketten an die aktuelle Position. Der einzige Sonderfall, der bearbeitet werden muss, sind Phrasen, welche in das Vorschaufenster laufen.

\[
\begin{array}{cccccccc}
0 & 0 & A & 0 & 0 & N & 6 & 3 & S \\
\end{array}
\]

\textit{Abbildung 2.23.: Datengröße der kodierter Nachricht des LZ77 Algorithmus}


Legt man die Suchfenstergröße und das Vorschaufenster so an, dass die Summe der Bits ein Vielfaches von 8 ist (wovon LZ77 ausgeht), kann man mit einem erhöhten Entropiewert rechnen, da es wahrscheinlich ist, dass einige Position-Längen-Kombinationen gehäuft vorkommen werden.

Fazit:
Der Algorithmus ist besonders speicherschonend, da kein separates Verzeichnis, Bäume oder andere Datenstrukturen angelegt werden müssen.
Allerdings wird beim Kodieren durch das Suchen der Phrase im Suchfenster verhältnismäßig viel Rechenleistung in Anspruch genommen.
2.3.2.2 LZSS


Neu ist die Einführung eines zusätzlichen Bits, welches angibt, ob es sich beim folgenden Kodierungszeichen um ein einzelnes Symbol der Eingangsnachricht handelt oder um eine Phrase. Bei einem einzelnen Symbol wird dann nicht wie bei LZ77 ein Tripel angegeben, sondern nur noch das Symbol an sich. Bei einer Phrase fällt zudem das angehängte Symbol weg, das sonst die Phrase ergänzt hat.


Nehmen wir bei dem Signal-Bit 0 für Einzelzeichen und 1 für Phrase an, so ergibt sich das Beispiel wie in Abbildung 2.24:

\[
\begin{array}{cccccccc|cc}
\text{Suchfenster} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\
\hline
\text{Vorschau} & A & N & A & S & 0 & A \\
\hline
\text{Ausgabe} & 0 & N & A & N & A \\
\end{array}
\]

\textit{Abbildung 2.24.: Kodierungsschritt 1 und 2 für LZSS}

Die beiden ersten Symbole werden wieder als Einzelsymbole kodiert; jeweils mit dem Signal-Bit 0 und dem Symbol an sich (Abbildung 2.24).

\[
\begin{array}{cccccccc|ccc}
\text{Suchfenster} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 1 & 2 & 3 \\
\hline
\text{Vorschau} & A & N & A & N & A & S \\
\hline
\text{Ausgabe} & 1 & 6 & 3 \\
\end{array}
\]

\textit{Abbildung 2.25.: Kodierungsschritt 3 für LZSS}

Bei Phrasen wird genauso vorgegangen wie beim LZ77 Algorithmus, nur dass die Ausgabe das Signal-Bit 1 vorangestellt und das Folgesymbol der Phrase nicht schreibt (Abbildung 2.25).
Abschließend noch einmal ein Einzelsymbol angeführt vom Signal-Bit 0 (Abbildung 2.26).

Zählt man alle Bits, die für die Kodierung der Eingangs Nachricht nötig sind, zusammen, so erhält man 33 Bit (Abbildung 2.27). Die gleiche Beispiel Nachricht durch LZB (siehe LZ77) kodiert, braucht 39 Bit. Dieser vergleichsweise große Unterschied entsteht vor allem durch die kürzere Kodierung von neuen Zeichen, die nicht im Suchfenster vorhanden sind.

Fazit:
Durch Bitwerte werden die Symbole im kodierten Datenstrom größtenteils so verschoben, dass sie sich über 2 Byte verteilen. Dadurch entsteht nicht mehr der erhöhte Entropiewert wie man ihn bei LZ77 erwarten kann. Als Einzelkodierung ist LZSS jedoch effizienter als LZ77, durch die deutlich kürzere Repräsentationsform von nicht im Suchfenster enthaltenen Symbolen.

2.3.2.3 LZ78

<table>
<thead>
<tr>
<th>Suchfenster</th>
<th>Vorschau</th>
<th>Ausgabe</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3</td>
<td>B S</td>
</tr>
<tr>
<td>A N A N A S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Abbildung 2.26.: Kodierungsschritt 4 für LZSS*

<table>
<thead>
<tr>
<th>Darstellungslänge</th>
<th>0 A 0 N 1 6 3 0 S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 8 1 8 1 3 2 1 8</td>
<td></td>
</tr>
</tbody>
</table>

*Abbildung 2.27.: Datengröße der kodierter Nachricht des LZSS Algorithmus*

<table>
<thead>
<tr>
<th>Index</th>
<th>Eintrag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>„Nullstring“</td>
</tr>
<tr>
<td>Restnachricht</td>
<td>Ausgabe</td>
</tr>
<tr>
<td>ANANAS</td>
<td>0,A</td>
</tr>
</tbody>
</table>

*Abbildung 2.28.: Kodierungsschritt 1 für LZ78*

Es kann kein Eintrag im Verzeichnis gefunden werden, der mit A beginnt, also wird in die Ausgabe der Index des „Nullstring“ als Escape-Zeichen gefolgt vom A geschrieben. Das A wird in das Verzeichnis als neuer Eintrag aufgenommen (Abbildung 2.28).

<table>
<thead>
<tr>
<th>Index</th>
<th>Eintrag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>„Nullstring“</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>Restnachricht</td>
<td>Ausgabe</td>
</tr>
<tr>
<td>NANAS</td>
<td>0,N</td>
</tr>
</tbody>
</table>

*Abbildung 2.29.: Kodierungsschritt 2 für LZ78*

Auch mit dem N am Anfang kann kein Eintrag gefunden werden, womit die Ausgabe auch hier mit dem Index 0 für den „Nullstring“ beginnt, gefolgt vom N. Das N wird ein neuer Eintrag im Verzeichnis (Abbildung 2.29).

<table>
<thead>
<tr>
<th>Index</th>
<th>Eintrag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>„Nullstring“</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
</tr>
<tr>
<td>Restnachricht</td>
<td>Ausgabe</td>
</tr>
<tr>
<td>ANAS</td>
<td>1,N</td>
</tr>
</tbody>
</table>

*Abbildung 2.30.: Kodierungsschritt 3 für LZ78*

Nun findet sich ein Eintrag im Verzeichnis, der mit A beginnt (Abbildung 2.30). Er hat zwar nur die Länge von einem Symbol, aber auch das ist ein Anfang. Der entsprechende
Index wird nun in die Ausgabe geschrieben. Angehängt wird das Symbol, was sich nach Abzug der im Verzeichnis gefundenen Phrase am Anfang der Restnachricht befindet. Im vorliegenden Fall ergibt dies das Tupel 1,N.
Anschließend wird das Verzeichnis noch um den neuen Eintrag ergänzt, der sich aus der gefundenen Phrase und dem folgenden Symbol ergibt. Hier ist dies der Eintrag AN.

<table>
<thead>
<tr>
<th>Index</th>
<th>Eintrag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>„Nullstring“</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
</tr>
<tr>
<td>3</td>
<td>AN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Restnachricht</th>
<th>Ausgabe</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>1,S</td>
</tr>
</tbody>
</table>

Abbildung 2.31.: Kodierungsschritt 4 für LZ78

Der nächste Schritt ist ziemlich gleich dem vorangegangenen. Auch hier wird der Index 1 mit dem Folgesymbol kombiniert. Das Ausgabetupel lautet 1,S (Abbildung 2.31).
Die Eingabenachricht ist damit komplett kodiert und es braucht nicht extra ein neuer VerzeichnisEintrag eingefügt zu werden.
Die Ausgabe sieht dann wie in folgender Abbildung 2.32 aus:

Abbildung 2.32.: Ausgabe der LZ78-Kodierung

Das Dekodieren verläuft im Aufbau des Verzeichnisses genauso wie beim Kodieren. Jedes dekodierte Tupel wird als neuer Eintrag in das Verzeichnis aufgenommen. Das Tupel wird dekodiert, indem die Zeichenkette am passenden Index, gefolgt vom angehängten Symbol, in die Ausgabenachricht geschrieben wird.
Optimiert man die ausgegeben Indexwerte so, dass sie nur die minimal nötige Bitlänge für die aktuelle Verzeichnisgröße haben, lässt sich die kodierte Nachricht auf 38 Bits verkleinern. Dies ist möglich, da auf beiden Seiten stets das gleiche Verzeichnis und somit die Indexzahl bekannt ist.

Fazit: Das Verzeichnis braucht zusätzlichen Speicher, kann dafür aber schneller durchsucht werden. Es werden sich mehr Zeichenketten gemerkt, als es beim „Sliding-Window“ möglich ist, was eine bessere Kodierung erlaubt.
2.3.2.4 LZW


![Abbildung 2.33: Eingangsnachricht für LZW](image)

Gegeben seien bis zu diesem Zeitpunkt auch folgende Einträge im Verzeichnis (Abbildung 2.34):

<table>
<thead>
<tr>
<th>Index</th>
<th>Zeichen</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>LZ7</td>
</tr>
<tr>
<td>257</td>
<td>LZ77</td>
</tr>
</tbody>
</table>

![Abbildung 2.34: Startverzeichnis für LZW](image)

wodurch die Symbole in der Ausgabe allerdings aus ihren Bytepositionen verschoben werden.

2.3.3 Entropiekodierung

Diese Kodierung ist eine Methode, die jedem einzelnen Zeichen eines Textes eine unterschiedlich gewichtete Repräsentation zuordnet. Sie steht im Gegensatz zu den Wortkodierungen, die eine ganze Folge von Zeichen des Originaltextes durch einen Index eines Verzeichnisses ersetzt.


Beispiel:

Eine uneindeutige Kodierung (Abbildung 2.35)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bitfolge</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>01</td>
</tr>
<tr>
<td>C</td>
<td>11</td>
</tr>
<tr>
<td>D</td>
<td>00</td>
</tr>
</tbody>
</table>

*Abbildung 2.35.: Nicht eindeutige Symbolkodierung*

<table>
<thead>
<tr>
<th>Nachricht</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beispielinterpretationen</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>D</td>
<td></td>
<td>C</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

*Abbildung 2.36.: Dekodierung einer Nachricht, dessen Kodierung nicht eindeutig ist*

Wie in der Abbildung 2.36 zu erkennen, sind mehrere Interpretationen der kodierten Nachricht möglich. Die ersten beiden lassen den Fehler nicht erkennen, die Dritte führt zu einem Fehler auf der Empfangsseite.

Ein Beispiel für eine eindeutige Kodierung findet sich in der Abbildung 2.37. Damit
kann jedes Symbol zweifelsfrei erkannt werden, sobald sein Kode vollständig vorliegt (Abbildung 2.38).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bitfolge</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>111</td>
</tr>
</tbody>
</table>

_Tabelle 2.37: Eindeutige Symbolkodierung_

2.3.3.1 Shannon-Fano-Kodierung

In Stichpunkten lässt sich der Algorithmus wie folgt zusammenfassen:

- Sortiere die Symbole nach ihrer Häufigkeit $n_i$,
- Teile die Symbole entlang dieser Reihenfolge so in 2 Gruppen, dass die Summe der Häufigkeiten in den beiden Gruppen möglichst gleich ist. Die beiden Gruppen entsprechen dem linken und rechten Ast unter dem Wurzelknoten des zu erstellenden Baumes,
- Befindet sich mehr als ein Symbol in einer der beiden entstandenen Gruppen, wende den Algorithmus rekursiv auf diese Gruppe an.

Am besten lässt sich dies natürlich an einem konkreten Beispiel erläutern. Folgendes
Wort bietet sich dabei an (Abbildung 2.39):

BETRIEBSSYSTEM

Abbildung 2.39.: Zu kodierende Nachricht für die Entropiekodierung

Nun wird die Häufigkeit der Symbole ermittelt und die Liste der Symbole danach sortiert (Abbildung 2.40).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>E</th>
<th>S</th>
<th>B</th>
<th>T</th>
<th>I</th>
<th>M</th>
<th>R</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_i )</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Abbildung 2.40.: Symbolhäufigkeiten sortiert

Wendet man nun den Algorithmus von Shannon und Fano an, so ergibt sich folgender Baum (Abbildung 2.41):

Abbildung 2.41.: Aufbau des Kodierungsbaums nach Shannon-Fano

Wesentlich ist hier die Tiefe der Blätter, da aus ihr jederzeit der gleiche sortierte binäre Baum rekonstruiert werden kann. Mit sortiertem Baum ist gemeint, dass links die Blätter mit der geringsten Tiefe stehen und die Tiefe der Blätter nach rechts hin ansteigt (Abbildung 2.42).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>E</th>
<th>S</th>
<th>B</th>
<th>T</th>
<th>I</th>
<th>M</th>
<th>R</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitlänge</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Abbildung 2.42.: Bitlänge der Symbole nach Shannon-Fano-Algorithmus

Um nun festzustellen, ob die gefundene Kodierung eine Verbesserung gegenüber der Ausgangsnachricht darstellt, werden die erhaltenen Daten anhand der Formeln
betrachtet. Dabei erhält man folgende Ergebnisse (Abbildung 2.43):

\[
\begin{align*}
H_s &= \text{mittlere Symbolaenge Shannon–Fano} \\
H_0 &= 3 \text{ Bit/Symbol} \\
H &= 2.8424 \text{ Bit/Symbol} \\
H_s &= 2.9286 \text{ Bit/Symbol} \\
R &= H_0 - H = 0.1576 \\
R_s &= H_s - H = 0.0862
\end{align*}
\]

Abbildung 2.43.: Vergleichsmaße der Kodierung aus dem Shannon-Fano-Algorithmus

Es ist zu erkennen, dass die Redundanz der Daten durch die neue Kodierung deutlich gesenkt worden ist. Die mittlere Kodewortlänge kommt der Entropie sehr nahe, lässt aber immer noch Raum für Verbesserungen.

Schließlich wird noch betrachtet, wie die Symbole anhand des binären Baumes kodiert werden. Dafür bekommt jede Kante einen binären Wert zugewiesen, also 0 oder 1. Im vorliegenden binären Graphen werden Kanten nach links mit einer 0 und Kanten nach rechts mit einer 1 gedeutet. Wird nun den Pfaden vom Wurzelknoten bis zu den Blättern gefolgt, ergeben sich folgende Symbolkodierungen (Abbildung 2.44):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>E</th>
<th>S</th>
<th>B</th>
<th>T</th>
<th>I</th>
<th>M</th>
<th>R</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kode</td>
<td>00</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>

Abbildung 2.44.: Symbolkodierungen aus dem Shannon-Fano-Algorithmus


Fazit:
Siehe hierzu Fazit-Huffman-Kodierung

2.3.3.2 Huffman-Kodierung
Der Shannon-Fano-Algorithmus zum Finden eines Kodebaums ist nicht in allen Fällen optimal. Aus diesem Grund schlug 1952 David A. Huffman einen verbesserten
Algorithmus vor, der nachweisbar immer einen optimalen Kodebaum hervorbringt [8]. Anstatt den Baum vom Wurzelknoten aus aufzubauen, beginnt dieser Algorithmus mit den Blättern. In Stichpunkten lässt sich der Vorgang des Kodebaum-Erstellens wie folgt beschreiben:


– Wiederhole, bis nur noch ein Baum übrig ist.

Am anschaulichsten lässt sich dies jedoch wieder am Beispiel erklären. Dazu wird auf das bereits bei der Shannon-Fano-Kodierung benutzte „BETRIEBSSYSTEM“ zurückgegriffen.

Die beiden Teilbäume werden abschließend zu einem Kodierungsbaum vereint. Es ist gut zu erkennen, wie Symbole mit gemeinsamen Häufigkeiten auf der selben Tiefe sind (Abbildung 2.47). Bei Shannon-Fano ist dies nicht der Fall.

Um nun festzustellen, inwieweit eine nachweisbare Verbesserung der Kodierung gegenüber dem Shannon-Fano-Algorithmus erreicht wurde, werden die Redundanzwerte bestimmt.

- 37 -

Fazit:

Im Vergleich mit der Shannon-Fano-Kodierung ergibt die Huffman-Kodierung stets einen optimalen Bitkode-Baum, ohne dabei weitere Nachteile mit sich zu bringen.

Als gemeinsames Problem erweist sich allerdings bei beiden voran behandelten Entropiekodierern die Übermittlung des Kodierungsbaumes, der die Größe der kodierten Daten gerade bei kleinen Nachrichten verhältnismäßig stark ansteigen lässt.

Unabhängig von ihrer Effizienz vereint die Shannon-Fano- und Huffman-Algorithmen ein Problem. Wie überträgt man den Kodierungsbaum an den Empfänger?


Man schreibe nun, von der Tiefe 1 beginnend, die Anzahl aller Blätter in der aktuellen
Tiefe und dahinter die Inhalte der Blätter. Man wiederhole den Vorgang dann solange mit der Menge aller Blätter der jeweils nächsten Tiefe, bis alle Blätter geschrieben wurden.
Für den Baum unseres Beispiels sieht das Ergebnis wie folgt aus:

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>E</th>
<th>S</th>
<th>2</th>
<th>B</th>
<th>T</th>
<th>4</th>
<th>I</th>
<th>M</th>
<th>R</th>
<th>Y</th>
</tr>
</thead>
</table>

*Abbildung 2.49.: Übermittelte Daten des Kodierungsbaum aus dem Huffman-Beispiel*

Auf der Empfängerseite kann der Baum anhand dieser Informationen wieder komplett rekonstruiert werden. Eine Angabe der Länge der Baumbeschreibung ist unnötig, da im Algorithmus erkannt werden kann, wann der Baum komplett ist.

### 2.3.3.3 Arithmetische Kodierung

Die Entropiekodierung, die über einen binären Kodierungsbaum die einzelnen Symbole in Bitfolgen übersetzt, ist eine einfach zu implementierende Lösung, die allerdings in bestimmten Fällen Schwächen zeigt. Die Kodierungslänge eines Symbols ist mindestens ein Bit, auch wenn die theoretische Kodewortlänge geringer ist. Weiterhin kann ein binärer Kodierungsbaum nur optimale Ergebnisse erzielen, wenn die Symbolwahrscheinlichkeiten jeweils eine negative Potenz von 2 sind. Die Blätter in einem binären Baum können nur Symbolwahrscheinlichkeiten von \(1 / (2^x)\) und kleiner 1 darstellen.
Bessere Ergebnisse verspricht die Arithmetische Kodierung. Dabei wird ein Intervall anhand der Symbolhäufigkeiten aufgeteilt. Ausgangsintervall ist dabei normalerweise 0 <= x < 1. Für die Kodierung eines Symbols wird dessen Teilintervall zur Basis für eine Neueinteilung. Auf diese Weise wird mit jedem kodierten Symbol das verfügbare Intervall kleiner. Ist die Nachricht komplett kodiert, kann anhand jedes Wertes aus dem


Im folgenden Beispiel wird die Nachricht "AABAAC" kodiert. Anhand der Symbolwahrscheinlichkeiten wird das Startintervall in die Teilintervalle 0 bis kleiner 4/6 für A, 4/6 bis kleiner 5/6 für B und 5/6 bis kleiner 1 für C eingeteilt. Das erste Symbol ist das A, was nach dem Algorithmus aus dem Teilintervall 0 bis kleiner 4/6 das Ausgangsintervall für den nächsten Schritt ist. Dieser wird wiederum anhand der Symbolwahrscheinlichkeiten unterteilt. Alle Schritte und die „Auffaltung“ der Teilintervalle sind in der Abbildung 2.50 zu sehen.

Die Abbildung 2.51 gibt einen Überblick über die Entwicklung der Teilintervalle. Eine Zeile stellt einen Kodierungsschritt dar und die Spalten die Teilintervalle der Zeichen.
Farbig unterlegt ist jeweils das Intervall, das für den nächsten Kodierungsschritt als Basis dient. Für das Beispiel erhält man am Ende ein Intervall von 0,32373 bis 0,3292. Das heißt jeder Wert innerhalb dieses Intervalls ermöglicht es, im Zusammenhang mit der Intervalleinteilung die Nachricht wieder herzustellen.

2.3.4 Unterstützende Verfahren


2.3.4.1 Burrows-Wheeler-Transformation

Der Algorithmus verändert die Reihenfolge der Eingabedaten so, dass sich die Wahrscheinlichkeit von Symbolwiederholungen erhöht. Das Prinzip beruht auf dem Umstand, dass bei bestimmten Symbolen die Wahrscheinlichkeit für gewisse Vorgängersymbole höher ist. So kommt in deutschen Texten vor einem „i“ recht häufig ein „e“[9]. M. Burrows und D. Wheeler haben ein Verfahren vorgestellt [9], das diesen Zusammenhang reversibel und platzsparend in die einfachere zu komprimierende Symbolwiederholung (siehe: Lauflängenkodierung Kapitel 2.3.1) und Ortsreduanzen (siehe: Move-To-Front-Transformation Kapitel 2.3.4.2) umwandeln kann.

Für die Transformation geht man folgendermaßen vor:

- Erstelle eine Liste aller möglichen Rotationen des Eingangsdatenstromes.
- Sortiere die Liste anhand der ersten Spalte (bei Gleichheit folgende Spalten).
- Schreibe die letzte Spalte in den Ausgang.

Dies soll an folgendem Beispiel demonstriert werden:

\textbf{ANANAS.}

\textit{Abbildung 2.52.: Eingangsdaten für Burrows-Wheeler-Transformation}

Die Liste der Rotationen stellt sich in einer Tabelle wie folgt dar (Abbildung 2.53):

<table>
<thead>
<tr>
<th>.</th>
<th>A</th>
<th>N</th>
<th>A</th>
<th>N</th>
<th>A</th>
<th>S</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

*Abbildung 2.53.: Liste aller Rotationen der Eingangsdaten vor der Sortierung*

Nun werden alle Zeilen anhand der Spalten sortiert, wobei die Spalten von links an gesehen die höhere Priorität besitzen (siehe Abbildung 2.54).

<table>
<thead>
<tr>
<th>A</th>
<th>N</th>
<th>A</th>
<th>N</th>
<th>A</th>
<th>S</th>
<th>.</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>N</td>
</tr>
<tr>
<td>A</td>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
</tr>
</tbody>
</table>

*Abbildung 2.54.: Liste aller Rotationen der Eingangsdaten nach der Sortierung*

Extrahiert man nun die letzte Spalte aus der Liste ergibt sich die Zeichenkette, wie man sie in Abbildung 2.55 sieht.

```
.NNAAA.A.S
```

*Abbildung 2.55.: Ausgangsdaten für Burrows-Wheeler-Transformation*

Schon in dieser recht kurzen Beispielzeichenkette lässt sich eine erhöhte Symbolwiederholung nach der Burrows-Wheeler-Transformation erkennen. Das einzige, was zusätzlich übermittelt werden muss, damit man die Ursprungs Nachricht wiederherstellen kann, ist der Index der richtigen Zeile in der Rotationstabelle nach der Sortierung. In unserem Fall wäre das die Zeile 7.
Allerdings stellt sich nun die Frage der Rücktransformation. Den dafür notwendigen Algorithmus kann man wie folgt kurz beschreiben:

- Schreibe die kodierte Nachricht abwärts in die letzte freie Spalte.
- Sortiere die Zeilen anhand der ersten gefüllten Spalte.
- Wenn noch nicht alle Spalten gefüllt sind, schiebe den Inhalt jeweils eine Spalte nach links und fahre mit dem ersten Punkt fort.
- Schreibe die Zeile mit dem übermittelten Zeilenindex in die Ausgabe.

Für die ersten Schritte soll dies nochmal am obigen Beispiel erläutert werden. Nachdem man die übermittelte Nachricht in die letzte Spalte eingetragen, sortiert und folgend um eine Spalte nach links verschoben hat, gleiches im zweiten Schritt nochmal geschehen ist, sieht die Tabelle wie folgt aus (Abbildung 2.56):

```
| . | A |
| N | A |
| N | A |
| A | N |
| A | N |
| A | S |
| . | . |
| S | . |
```

Abbildung 2.56.: Rücktransformation nach Burrows-Wheeler im ersten Schritt

```
| . | A | N |
| N | A | N |
| N | A | S |
| A | N | A |
| A | N | A |
| A | S | . |
| . | . | A |
| S | . | . |
```

Abbildung 2.57.: Rücktransformation nach Burrows-Wheeler im zweiten Schritt
Noch lässt sich in der Tabelle nicht wirklich etwas erkennen, was sich mit dem Fortschreiten des Dekodierens allerdings ändert (Abbildung 2.57).

Inzwischen sind in den letzten drei Spalten deutlich Fragmente zu identifizieren, die aus der Nachrichtenzeichenkette stammen, wie sie der Burrows-Wheeler-Transformation als Eingangsdaten dienten. Die folgenden Schritte werden aus Platzgründen übersprungen, um gleich auf die abschließende Tabelle zu kommen (Abbildung 2.58).

Die Tabelle repräsentiert die Rotationstabelle vor der Übertragung. Der übermittelte Zeilenindex 7 deutet nun korrekt auf die Ursprungsrichtung.


---

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>N</th>
<th>A</th>
<th>N</th>
<th>A</th>
<th>S</th>
<th>.</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>N</td>
</tr>
<tr>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>S</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>S</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>N</td>
<td>A</td>
<td>A</td>
<td>S</td>
</tr>
</tbody>
</table>

*Abbildung 2.58.: Rücktransformation nach Burrows-Wheeler im dritten Schritt*

*Abbildung 2.59.: Rücktransformation nach Burrows-Wheeler im letzten Schritt*
2.3.4.2 Move-To-Front-Transformation

Die Symbolverteilung kann in unterschiedlichen Teilen einer Nachricht völlig verschieden sein. Ein Extrembeispiel wäre eine Nachricht, deren erste Hälfte nur aus Zeichen von 0 bis 127 besteht, die andere Hälfte dagegen nur aus den Zeichen 128 bis 255. Für die komplette Nachricht mag die Auftrittswahrscheinlichkeit für alle Symbole gleich sein, aber betrachtet man die beiden Hälften getrennt, entsteht ein ganz anderes Bild.

Die bekannten Komprimierungsverfahren auf solche Ortsredundanzen anzupassen, ist eine Möglichkeit, allerdings ist das alles andere als trivial und verkompliziert die Algorithmen in hohem Maße.

Einen überraschend einfachen und sehr performanten Ansatz liefern wiederum M. Burrows und D. Wheeler in ihrer Arbeit. Dieser überführt die Ortsredundanzen in günstigere Symbolwahrscheinlichkeiten für die komplette Nachricht. Dabei passt man lediglich das Alphabet ständig an die aktuellen Umstände an. Es gilt folgende einfache Regel:

- Entferne das zuletzt aufgetretene Zeichen aus dem Alphabet und füge es an der ersten Position wieder ein.

In die Ausgabe wird immer der Index des aktuellen Zeichens im Alphabet geschrieben. Als Beispielieingabe soll die transformierte Nachricht der oben dargelegten Burrows-Wheeler-Transformation dienen. Als Alphabet nehmen wir nur die Menge der Symbole im Eingabestring an, damit das Beispiel übersichtlich bleibt und bei der kurzen Eingabelänge eine erkennbare Wirkung erzielt.

In der ersten Spalte stehen die Eingabedaten, wobei das aktuelle Symbol markiert ist. Die zweite und dritte Spalte stellen das aktuelle Alphabet und die aktuellen Ausgabedaten dar. Jede Zeile repräsentiert einen Arbeitsschritt (Abbildung 2.60).
<table>
<thead>
<tr>
<th>Eingabedaten</th>
<th>Alphabet</th>
<th>Ausgabedaten</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NNAAAA.S</td>
<td>ANS.</td>
<td>4</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>.ANS</td>
<td>43</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>N.AS</td>
<td>431</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>N.AS</td>
<td>4313</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>AN.S</td>
<td>43131</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>AN.S</td>
<td>431311</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>AN.S</td>
<td>4313113</td>
</tr>
<tr>
<td>.NNAAAA.S</td>
<td>.ANS</td>
<td>43131134</td>
</tr>
</tbody>
</table>

*Abbildung 2.60.: Beispiel einer Move-To-Front-Transformation*

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Entropie vor MoveToFront</th>
<th>Entropie nach MoveToFront</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( H ) = 1,9056 Bit/Symbol</td>
<td>( H_c ) = 1,5613 Bit/Symbol</td>
</tr>
</tbody>
</table>

Abbildung 2.61: Vergleich der Entropie vor und nach der Move-To-Front-Transformation

Beim Vergleich der Entropie beider Zeichenketten stellt man fest, dass diese nach der Move-To-Front-Transformation, bei gleicher Nachrichtenlänge, deutlich niedriger ist (Abbildung 2.61). Das gibt Entropiekodierungen wie dem Huffman-Verfahren eine viel günstigere Ausgangssituation, was bessere Ergebnisse erwarten lässt.


2.4 Kombination der Komprimierungsalgorithmen

2.5 Eignung der Algorithmen entsprechend der Anforderungen der Arbeit

Bei der Auswahl der Algorithmen für die vorliegende Arbeit dürfen die Limitierungen der eingebetteten Systeme und die kleine Eingangsdatenmenge der zu komprimierenden Pakete nicht außer Acht gelassen werden. Ziel ist es unter anderem Strom zu sparen durch weniger Datenverkehr bei minimalem zusätzlichen Rechenaufwand. In Abbildung 2.64 wird eine kurze Übersicht über die behandelten Verfahren dieses Kapitels und darüber, welcher Rechenaufwand auf eingebetteten Systemen zu erwarten ist, gegeben. Daneben ist eine Einschätzung der erwarteten Zusatzinformationen, die die Verfahren in den kodierten Daten benötigen. Die Eingangsdatenmenge wird zwischen 10 und 100 Byte angenommen.
Die beiden entropiekodierenden Verfahren verlangen im Normalfall die Übermittlung der Symbolwahrscheinlichkeiten, aus denen sich die Kodierung ableitet. Diese können in ungünstigen Fällen größer als die kodierte Nachricht sein. BWT erfordert auf beiden Seiten das Sortieren von Daten, die dem Umfang der Eingangsdaten entspricht. LZ77 und LZSS benötigen viel Zusatzinformationen, um eine Phrase oder ein Symbol zu kodieren. Dies können sie nur durch lange wiederholte Phrasen ausgleichen, die in so kleinen Datenmengen unwahrscheinlich sind. LZ78 hat ähnliche Probleme, insbesondere beim „Warmlaufen“ des Verzeichnisses mit Einzelsymbolen, was durch das vorinitialisierte Verzeichnis in LZW ausgeglichen werden kann. BWT benötigt unabhängig von der Eingangsdatenmenge nur einen Index für die Rücktransformation. MTF kommt völlig ohne Zusatzdaten aus.


<table>
<thead>
<tr>
<th>Algorithmus</th>
<th>Rechenaufwand</th>
<th>Zusatzinformationen bei kleinen Eingangsdaten</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lauflängenkodierung</td>
<td>gering</td>
<td>gering</td>
</tr>
<tr>
<td>LZ77</td>
<td>hoch</td>
<td>hoch</td>
</tr>
<tr>
<td>LZSS</td>
<td>hoch</td>
<td>hoch</td>
</tr>
<tr>
<td>LZ78</td>
<td>hoch</td>
<td>hoch</td>
</tr>
<tr>
<td>LZW</td>
<td>hoch</td>
<td>durchschnittlich</td>
</tr>
<tr>
<td>Huffmankodierer</td>
<td>sehr hoch</td>
<td>sehr hoch</td>
</tr>
<tr>
<td>Arithmetischer Kodierer</td>
<td>sehr hoch</td>
<td>sehr hoch</td>
</tr>
<tr>
<td>BWT</td>
<td>durchschnittlich</td>
<td>sehr gering</td>
</tr>
<tr>
<td>MTF</td>
<td>gering</td>
<td>keine</td>
</tr>
</tbody>
</table>

*Abbildung 2.64: Rechenaufwand und Zusatzinformationenanteil der Komprimierungsalgorithmen*
2.6 Verfahren zur Aggregation

Die Aggregation erfordert weniger komplexe Algorithmen. Hierbei werden nur die einkommenden Pakete zu einem großen Paket verschmolzen und müssen danach wieder sauber trennbar sein.

Dies kann erreicht werden, indem an den Anfang des Pakets eine Art Verzeichnis legt mit der Länge oder Position aller enthaltenen Daten. Da aber eigentlich nur immer ein Paket nach dem anderen aus dem großen Paket wieder herausgenommen wird, ist es in der Verarbeitung angenehmer, wenn die nötigen Daten zum Entpacken des Teilpakets immer einfach und direkt verfügbar sind. Wenn beim Verschmelzen also einfach nur die Paketdaten gefolgt von der Paketlänge auf den Stapel gelegt werden, so ist bei der Trennung vom Paket die Paketlänge zur Hand. Auf diese Weise lassen sich gezielt die Daten des zu behandelnden Teilpakets entnehmen.
3 Entwurf einer Aggregationsschicht für COPRA

3.1 Rahmenwerk der Arbeit

Der folgende Entwurf ist abhängig vom gegebenen Rahmenwerk, in welchem er umgesetzt werden soll. Für dessen Verständnis späterer Überlegungen folgt eine kurze Einführung in die wichtigsten Begriffe.

3.1.1 REFLEX

REFLEX ist ein generisches, ereignisgesteuertes Betriebssystem für tief eingebettete Systeme. Kontroll- und Steuerfunktionen werden durch passive Objekte repräsentiert, die gemäß einer wählbaren Schedulingstrategie, wie zum Beispiel EDF (Earliest Deadline First), präemptiv aktiviert werden. Die Kommunikation zwischen den Objekten erfolgt über ein Ereignisflussmodell, das einem Datenflussmodell sehr ähnlich ist.

3.1.2 COPRA


3.1.3 SERNET

Die Arbeit und Kommunikation der Knoten lässt sich mit dem SERNET-Emulator

3.2 Planung der Umsetzung

Ziel dieser Arbeit ist es, eine Aggregations- und Komprimierungsschicht für COPRA zur Verfügung zu stellen. Daher sind folgende Überlegungen auch an COPRAs Rahmenbedingungen geknüpft. Aus Sicht der zu implementierenden Schicht sind folgende Ausgangsbedingungen gegeben:

- Es gibt jeweils eine Transfer-Richtung (Senden und Empfangen) in der Pakete über eine deliver()/accept()-Methode zugestellt werden und durch eine txForward()/rxForward()-Methode weitergeleitet werden.

- Pakete können dynamisch aus einem Pool heraus erzeugt werden.

- Ein Paket kann von anderen Schichten weiter gehalten werden und es darf deshalb nicht einfach gelöscht oder seine Daten manipuliert werden.

Bei der Umsetzung der Algorithmen ist darauf zu achten, dass man über keinen dynamischen Speicher verfügt und Fließkommaoperationen auf eingebetteten Systemen nur mit hohem Aufwand realisierbar sind. Beides müsste über zusätzliche Bibliotheken bereitgestellt werden, was zu höherer Kodegröße führt. Gerade bei Verzeichnissen, zum Beispiel für Wortkodierer und Kodierungsbäume, würden diese Einschränkungen zusätzlichen Aufwand bedeuten. Die Standard-Datentypen und Algorithmen der C/C++ Bibliotheken stehen ebenfalls nicht zur Verfügung. Aufgrund der Analyse kann eigentlich nur die Aggregation auf ein Verfahren
eingegrenzt werden. Einzig ein optionaler Timer wird integriert, der nach einer bestimmten Zeit die bisher gesammelten Pakete an die nächste Schicht weitergibt.


3.3 Vorhalten sendebereiter Pakete

Damit die aggregierten Daten von außen ausgelöst an die nächste Schicht durchgereicht werden können, sollen sie stets in einem sendebereiten Zustand zur Verfügung stehen. Löst zum Beispiel die MAC-Schicht die Weiterleitung aus, da in kurzer Zeit ein Senderahmen zur Verfügung steht, würde eine längere Paketbearbeitung zum Verpassen des Senderahmens führen. Daraus ergibt sich die Notwendigkeit, immer dieses fertige Paket im Speicher zu halten.

Weiterer Speicher wird bei der Komprimierung in Kombination mit der Aggregation in Anspruch genommen. Das separate Komprimieren von Teildaten einer Nachricht führt im allgemeinen Fall zu schlechteren Ergebnissen, als wenn die verschmolzene Nachricht komprimiert wird. Daher muss man die verschmolzenen Nachrichten der Aggregation unkomprimiert verfügbar halten, um sie nach jeder neu aggregierten Teilnachricht neu zu kodieren. Dies durch separates Entpacken für die Aggregation gefolgt vom Neukomprimieren zu umgehen, kommt aufgrund des Rechenaufwands für diese Arbeit nicht in Frage.

3.4 Maße der Praxistauglichkeit


3.5 Integration ins Rahmenwerk

Die Komprimierung und Aggregation wird ins Rahmenwerk als eine PPS eingebunden. Sie sollte dabei möglichst tief in der Abfolge der PPE eingesetzt werden, damit

Als Klassenname wurde „Compression“ für die kombinierte Schicht von Aggregation und Komprimierung gewählt (Abbildung 3.2).

### 3.6 Interne Aufbau der Compression-Schicht

Wird die Compression-Schicht mit ihrem vollem Funktionsumfang konfiguriert, entspricht sie dem folgenden Diagramm (Abbildung):
Die Aggregation verschmilzt solange Pakete aus der übergeordneten Schicht, bis diese einen Schwellwert beim Datenvolumen erreicht haben, oder der Timer ein Signal gibt. Dann wird das neue Paket, das alle aggregierten Teilpakete und die Informationen, wie sie wieder sauber trennbar sind, enthält, an die Komprimierung weitergegeben. An diesem Punkt durchläuft das Paket alle Komprimierungsalgorithmen, welche per Konfiguration vorgegeben werden. Anschließend folgt die Übergabe an die nächste Schicht in der PPE.

Per Konfiguration ist es möglich, alle Teilabläufe anzupassen oder zu entfernen. So sind zum Beispiel Aggregationen ohne Timer möglich, eine leere Komprimierungsphase oder eine Komprimierung ganz ohne Aggregation.
4 Implementierung


4.1 Implementierung der Komprimierungsalgorithmen

Die Basisklasse aller Komprimierungsalgorithmen bildet CompressorCodec. Diese bietet nach außen die nötigen Schnittstellen encode() und decode() an. Ihr grundlegender Inhalt ist der Abbildung 4.1 zu entnehmen.

```
Typedef UINTMAX unsigned int;
typedef BYTE unsigned char;

class CompressorCodec
{
    public:
        virtual UINTMAX encode(BYTE* inputBuffer, UINTMAX inputBufferSize, BYTE* outputBuffer, UINTMAX outputBufferSize) = 0;
        virtual UINTMAX decode(BYTE* inputBuffer, UINTMAX inputBufferSize, BYTE* outputBuffer, UINTMAX outputBufferSize) = 0;
    }
```

_Abbildung 4.1.: Klassendeklaration des Interface CompressCodec_

Die Lauflängenkodierung ist am problemlosesten umzusetzen, da sie kein Verzeichnis und keine Kodierungstabelle zu halten braucht. Im Rahmen der Arbeit fand neben der einfachen Variante mit Escape-Zeichen (_RleCodec_) auch gleich die PCX-Variante

Als Vertreter der Entropiekodierer wurde die Huffman-Kodierung gewählt, da sie die problemloseste Umsetzung verspricht. Trotzdem ist sie im Vergleich zu den bisher im Rahmen dieser Bachelorarbeit implementierten Komprimierungsalgorithmen jene, die die komplexeste Implementierung nach sich zieht. Der binäre Baum, welcher die Bit-Kodierungen der Symbole enthält, muss ebenfalls in einem flachen Feld gespeichert werden, da die dynamischen Mechanismen des Speichers fehlen. Im Gegensatz zum Verzeichnis (Dictionary) und dem Bit-Feld (BitField) wurde dies allerdings nicht in einer eigenen Klasse realisiert, um Optimierungen, speziell auf die Huffman-Kodierung angepasst, vornehmen zu können. Daraus resultieren allerdings komplexere Abhängigkeiten der Teilklassen, welche in Abbildung 4.4 dargestellt sind. Neben der eigentlichen Kodiererklasse HuffmanCodec werden CharNode zum ersten Aufbau des Kodierungsbaumes, CharCode als Repräsentant des Bitkodes für ein Symbol und CharCodeNode für die Arbeit mit dem Kodierungsbaum verwendet.

MergeSort erstellt, in der Vermutung, dass das Sortieren ein häufiger auftretendes Problem darstellt.

4.2 Einbindung der Komprimierungsalgorithmen in die Netzwerkschicht

Da das von der Vorgängerschicht übergebene Paket nicht verändert werden darf, werden die kodierten Daten in einem Puffer auf dem Systemstack abgelegt und abschließend ein neues Paket erzeugt, das die Daten des Puffers übernimmt.
Dabei wird die direkte Behandlung eines Pakets nicht unterbrochen. Wird oben ein Paket an die Schicht übergeben, kommt nach der Komprimierung unten ein Paket wieder heraus. Bis auf die Zeit, welche zur Kodierung der Nachricht gebraucht wird, ist diese Stufe der Umsetzung voll transparent.

4.3 Implementierung der Aggregation

Mit der Aggregation kommen tiefere Eingriffe in die Arbeit des Rahmenwerks. Wird ein Paket oben in die Aggregationsschicht hinein gegeben, so heißt dies nicht mehr, dass

4.4 Zusammenspiel von Aggregation und Komprimierung

Damit mehr logische Pakete in ein physikalisches Paket passen, werden die aggregierten Daten komprimiert. Das ermöglicht in vielen Fällen das zusätzliche Hinzufügen von Paketen, was die Effizienz der Aggregation erhöht. Umgekehrt wächst die Effizienz der Komprimierung mit der Größe der Eingangsdaten. Erst durch die Verschmelzung von Einzelpaketen wird die Komprimierung wirklich erfolgversprechend.

4.5 Optimierungen in der Implementierung

Die Paketaggregation wurde so überarbeitet, dass nur versucht wird, die Daten zu komprimieren, wenn sie größer als die maximale Paketgröße für ausgehende Pakete sind. Solange die Daten unkomprimiert in ein Paket passen, werden sie nur aggregiert. Weiterhin wäre die Umsetzung einer Komprimierung denkbar, die ihr Phrasenverzeichnis oder die Symbolverteilung statisch festsetzt. Damit ließe sich Rechenaufwand sparen, der sonst bei jedem Komprimierungsdurchlauf nötig ist, um diese zu erstellen. Bei den Entropiekodierern ließe sich dadurch auch eine Verringerung der zu übertragenden Daten erreichen. Bedingt ist diese Überlegung allerdings durch eine genaue Kenntnis der Paketdaten, welche an die Komprimierung übergeben werden. Dies lässt sich nur separat für jeden praktischen Fall ermitteln.

4.6 Beispielkonfiguration der Compression-Schicht

Das Einbinden der Compression-Schicht findet äquivalent dem anderer COPRA-
Schichten statt. Über die Methoden `txConnect()` und `rxConnect()` werden die direkten Nachbarn in der PPE-Kette festgelegt. Der NetBuffer-Pool wird mit dem Konstruktor übergeben (Abbildung 4.5).

```cpp
#include "copra/pps/Compression.h"

class RoutingTransportCompPPE
{
public:
    TinyCompPPE([...]) : [...], compression(pool)
    {
        reliability.txConnect(&compression);
        compression.txConnect(&txCRC);
        txCRC.txConnect(ppeEnd);
        
        rxConnect(&rxCRC);
        rxCRC.rxConnect(&compression);
        compression.rxConnect(&reliability);
    }
    [...]
    Compression compression;
    [...]
};
```

*Abbildung 4.5.: Einbindung der Compression-Schicht in eine PPE*

Konfiguriert wird die Compression-Schicht in der Datei CompressConfig.h. Primär kann hier festgelegt werden, welche Teilabläufe aktiviert sind und welche nicht benutzt werden sollen. `COMPRESS_MERGE_ON` aktiviert die Aggregation und kann durch `COMPRESS_TIMEOUT_ON` durch den Timer ergänzt werden. Unabhängig davon kann mit `COMPRESS_COMPRESS_ON` die Komprimierung aktiviert werden (Abbildung 4.6).

```cpp
#ifndef COMPRESS_MERGE_ON
#define COMPRESS_MERGE_ON
#endif

#ifndef COMPRESS_TIMEOUT_ON
#define COMPRESS_TIMEOUT_ON
#endif

#ifndef COMPRESS_COMPRESS_ON
#define COMPRESS_COMPRESS_ON
#endif
```

*Abbildung 4.6.: Basiskonfiguration in der CompressConfig.h*
Zusätzlich lassen sich wichtige Schwellwerte einstellen (Abbildung 4.7).  
**COMPRESS_BUFFER_MAXSIZE** ist die maximale Größe, welche alle aggregierten Teilpakete inklusive Zusatzinformationen unkomprimiert belegen dürfen. Wird dieser Wert überschritten kommt es zu einem Flush.

**COMPRESS_BUFFER_PACKETSIZE** beschreibt die maximale Größe, die ein fertig verschmolzenes und/oder komprimiertes Paket einnehmen darf, damit es an die nächste Schicht weitergegeben werden kann.

Mit **COMPRESS_TIMEOUT_FLUSH** wird angegeben, wie viel Zeitzyklen maximal auf weitere einkommende Pakete zur Aggregation gewartet wird. Ist dieser Wert erreicht, kommt es zum Flush.

```c
enum {
    COMPRESS_BUFFER_MAXSIZE      = (PacketPayload * 2),
    COMPRESS_BUFFER_PACKETSIZE   = (PacketPayload - 20),
    COMPRESS_TIMEOUT_FLUSH       = 10
};
```

*Abbildung 4.7.: Detailkonfiguration in der CompressConfig.h*

```c
#ifdef COMPRESS_COMPRESS_ON
    #include "compress/rle2/Rle2Codec.h"
    #include "compress/lzw/LzwCodec.h"

    #define COMPRESS_CODEC_COUNT 2

    #define COMPRESS_CODEC_CODECS
    Rle2Codec codec0; \
    LzwCodec codec1; \
    CompressorCodec* codecs[COMPRESS_CODEC_COUNT]; \
    \
    codecs[0] = &codec0;\n    codecs[1] = &codec1;
#endif //COMPRESS_COMPRESS_ON
```

*Abbildung 4.8.: Komprimierungskonfiguration in der CompressConfig.h*
5 Bewertung


In der Praxis erfüllen Sensornetzwerke Aufgaben, wie die schon erwähnte Überwachung eines Areals. Dabei werden Daten gesammelt, die zu einer Senke hin „abfließen“. In solchen Anwendungsszenarien kann es zu einem Flaschenhals auf dem Weg zum Zielknoten kommen, was im Zusammenhang mit Timeouts zu Paketwiederholungen führt. Ein weiteres denkbares Szenario ist Kommunikation aller Knoten miteinander, das heißt, jeder Knoten kann Sender und Empfänger eines Anwendungspaketes sein. Hier teilen sich Pakete auf ihrem Weg kürzere Pfade. Es ist häufiger der Fall, dass sich in einem Knoten Teilpfade kreuzen. Denkbar ist solch eine Kommunikation in einem Sensornetz, das sich selbst organisiert. Für die Simulationen kommt eine Anwendung zum Einsatz, die beide Kommunikationsarten zeitlich ineinander übergehend verwendet.

5.1 Aufbau und Ablauf der Simulationen in SERNET

Bei den Simulationen kamen folgende Konfigurationen zum Einsatz:

1. Funktionsleere Schicht: Referenzwerte, um Schlüsse auf die Ausgangslage zu ziehen


4. Aggregation Timeout 1, Komprimierung RLE – LZW: Aggregation mit einem Zeitzyklus als maximale Sammeldauer und aktivierter Komprimierung

5. Aggregation Timeout 2, Komprimierung RLE – LZW: Aggregation mit zwei Zeitzyklen als maximale Sammeldauer und aktivierter Komprimierung

6. Aggregation Timeout 4, Komprimierung RLE – LZW: Aggregation mit vier Zeitzyklen als maximale Sammeldauer und aktivierter Komprimierung

7. Aggregation Timeout 1: Aggregation mit einem Zeitzyklus als maximale Sammeldauer

8. Aggregation Timeout 2: Aggregation mit zwei Zeitzyklen als maximale Sammeldauer

9. Aggregation Timeout 4: Aggregation mit vier Zeitzyklen als maximale Sammeldauer


Folgende Anwendungsszenarien kamen zum Einsatz (Abbildung 5.1 und 5.2):
Abbildung 5.1 stellt eine Ringanordnung dar. Der Knoten mit der ID 0 liegt außerhalb dieses Ringes, in der Abbildung unten links. Dieser Netzaufbau wird im Folgenden als „6nodes“ bezeichnet.

In der Abbildung 5.2 ist der Netzaufbau für „line6“ zu sehen. Hier bilden alle Knoten eine Kette, in der jeder Knoten nur zu seinen direkten Nachbarn Funkkontakt hat.


Folgende, für den Testfall wichtige Werte wurden gesetzt:

- Frequenz der Pinganfragen: alle 60 Zeitzyklen
- Maximale Paketgröße im Radio: 65 Byte

- Maximale Paketgröße, mit welcher die Aggregations- und Komprimierungsschicht Pakete an die nächste (Richtung Radio) Schicht weiter gibt: 45 Byte

- Zeitzyklus, mit dem die Timer betrieben werden: 250 Millisekunden


### 5.1.1 Betrachtung der Paketanzahl pro Anwendungsdurchlauf

Die Testanwendung brachte es in Durchläufen der Version 1 (keine Aggregation und keine Komprimierung) im Netzwerkaufbau line6, welches eine Linie von Netzknoten abbildet, auf die in der Tabelle dargestellten Referenzwerte (Abbildung 5.3, komplette Messtabelle siehe Abbildung 5.15).

<table>
<thead>
<tr>
<th>gesendete Pakete</th>
<th>566</th>
</tr>
</thead>
<tbody>
<tr>
<td>gesendete Byte</td>
<td>11.606 Byte</td>
</tr>
<tr>
<td>durchschnittliche Paketgröße</td>
<td>20,5 Byte</td>
</tr>
</tbody>
</table>

*Abbildung 5.3.: Referenzmessung für line6*

Die Paketgröße von knapp 21 Byte und der Umstand, dass alle Routen ein Teilpfad von Knoten 0 zu 5 oder andersrum sind, lassen davon ausgehen, dass durch Aggregation die Paketanzahl spürbar gesenkt werden kann.

In den folgenden Diagrammen stellt das gefüllte Quadrat stets den Wert vor der Aggregations- und Komprimierungsschicht dar und das leere Quadrat den Wert, wie er von der darunter liegenden COPRA-Schicht gesehen wird. Diese Werte werden immer auf der Sendeseite aller Knoten gemessen. Da es nur einen Übertragungskanal gibt, ist dies ausreichend.
In Abbildung 5.4 lässt sich erkennen, dass die Programmversionen ohne Aggregation (2, 3) nur wenig Einfluss auf die Paketanzahl hatten. Durch reine Komprimierung konnten nur maximal 10% der Pakete eingespart werden. Mit aktivierter Aggregation ist die Anzahl der ausgehenden Pakete überraschend schwach gesunken (Programmkonfiguration 7 und 8) oder sogar gegenüber der Referenzmessung gestiegen (Programmkonfiguration 9). Die Steigerung der eingehenden Pakete (Pakete, die der Aggregation übergeben werden) lässt sich in Version 8 und 9 mit der Verzögerung erklären, die die Aggregation verursacht, während sie auf weitere Pakete wartet. In dieser Zeit laufen auch Limits anderer Schichten aus, was zur Sendewiederholung führt. Die geringe Minimierung der ausgehenden Pakete ist zu erklären mit den für diesen Testfall ungünstigen Rahmenwerten. Ohne die Aggregation sind die Pakete circa 21 Byte groß. Die maximale Paketausgangsgröße an untere Schichten ist für die Aggregation allerdings mit 45 Byte festgesetzt. Zwei eingehende Pakete inklusive Zusatzinformationen der Aggregationsschicht durchbrechen diesen Schwellwert sehr oft, was dazu führt, dass die Pakete trotzdem häufig allein verschickt werden müssen. Die Komprimierung hebt diesen Nachteil offensichtlich auf. Die Paketanzahl ist deutlich gesunken. Beste Ergebnisse erhält man mit ein oder zwei Wartezyklen (4, 5), wohingegen bei vier Wartezyklen bereits die oben beschriebene Anzahl der Anwendungspakete negativ beeinflusst wird. Die Programmversion 5 mit durchschnittlich 199 ausgegebenen Paketen stellt eine Reduktion der Anwendungspaketanzahl auf knapp 65% dar. Vergleicht man diesen Wert mit der
Referenzmessung (1), konnte die Paketanzahl auf 35% reduziert werden.

Der Referenzdurchlauf mit 6nodes ergab folgende Referenzwerte (Abbildung 5.5, komplette Messtabelle siehe Abbildung 5.16):

<table>
<thead>
<tr>
<th>gesendete Pakete</th>
<th>274</th>
</tr>
</thead>
<tbody>
<tr>
<td>gesendete Byte</td>
<td>5.140 Byte</td>
</tr>
<tr>
<td>durchschnittliche Paketgröße</td>
<td>18,8 Byte</td>
</tr>
</tbody>
</table>

*Abbildung 5.5.: Referenzmessung für 6nodes*

Auch in der Simulation mit dem Anwendungsszenario „6nodes“ zeigen die ersten drei Programmkonfigurationen keine wirklichen Abweichungen (siehe Abbildung 5.6). Die geringe Paketgröße der Referenzmessung lässt aber darauf schließen, dass die Aggregation diesmal erfolgreicher sein wird. Inklusive Zusatzdaten der Aggregationsschicht passen viel häufiger zwei ausgehende Pakete in ein zu sendendes Paket. Dies zeichnet sich auch gut erkennbar in den gemessenen Werten (7, 8, 9) ab. Diese sind durchweg besser als der Referenzwert. Bei zwei Wartezyklen wird eine Minimierung der Paketanzahl um immerhin knapp 40% erreicht. Dies reicht schon fast an die Werte der Aggregation inklusive Komprimierung heran (4, 5, 6), die bis zu 58% erreichen. Der direkte Vergleich der Programmversion 5 zum Referenzwert ergibt sogar 42%.

*Abbildung 5.6.: Paketanzahl für Testfall 6nodes*
5.1.2 Betrachtung des gesendeten Datenvolumens

Ein zweiter wichtiger Messwert neben der Paketanzahl ist das während des Anwendungsdurchlaufs auftretende Datenvolumen.

Die Abbildung 5.7 stellt die Ergebnisse der Simulation mit line6 dar. Da bei der Aggregation nur Zusatzinformationen (Länge des Teilpakets) der Aggregationsschicht hinzukommen, aber keine entfernt oder komprimiert werden, ist die Zunahme des Datenvolumens in diesen Fällen (7, 8, 9) wenig überraschend. Bei reiner Komprimierung fällt das Datenvolumen, wenn der richtige Komprimierungsalgorithmus gewählt wurde, wie es mit RLE-LZW (3) der Fall zu sein scheint. BWT-RLE (2) vergrößert dagegen das Datenvolumen im Vergleich zu den Eingangsdaten. Bei kombinierter Aggregation und Komprimierung wird das Datenvolumen im Vergleich zur theoretischen Größe noch einmal merkbar verkleinert, wobei die Verbesserung im Vergleich zum Referenzwert vor allem durch die gesenkte Paketanzahl zustande kommt.
Ein ähnliches Bild zeigt sich beim Testfall 6nodes (Abbildung 5.8). Die reine Aggregation (7, 8, 9) lässt das Datenvolumen ansteigen. Dadurch bleibt lediglich die Konfiguration 8 unter dem Datenvolumen der Referenzmessung, begünstigt durch die geringere Paketanzahl.

5.1.3 Detaillierte theoretische Betrachtung ausgesuchter Experimente

Die Programmkonfigurationen 5 und 8 werden näher untersucht, da sie in den gewählten Netzsituationen vergleichsweise gute Ergebnisse bezüglich der Paketanzahl und des Datenvolumens zeigten.

Geht man vom Testszenario 6nodes aus, ergeben sich für die Referenzmessung folgende Realwerte. 5140 gesendete Byte mal 1,25 (2 Zusatzbits je Zeichen) ergibt 6.425 real gesendete Byte. Bei 274 Paketen kommt man auf eine Präambelsumme, die vergleichbar ist mit 8694 übertragenen Byte (inklusive Zusatzbits). In der Summe sind das 15.119 Byte, die über die Antenne gesendet wurden.

Bei der Programmversion 8 ergab sich bei der Messung eine Paketanzahl von 166 bei gesamt 4.688 übertragenen Byte. Nach obigen Rechnungen entspricht das 11.127 real übertragenen Byte, was einer Reduktion gegenüber der Referenzmessung von circa 26% entspricht.

Für die Programmkonfigurationen 5 sind das bei 114 Paketen und 3.811 übertragenen Byte insgesamt 8.381 real übertragene Byte. Das entspricht annähernd einer Halbierung der real übertragenen Daten und somit auch der für die reine Übertragung aufgewendeten Energie.

Laut der Überlegung, dass man für ein übertragenes Byte im Gegenzug 1.000 Byte berechnen kann, würde dies für die Programmkonfigurationen 8 circa 3.992.000 Operationen bedeuten. In diesem Testlauf wurden 221 Pakete an die Aggregationsschicht übergeben, was pro Aggregationsdurchlauf circa 18.000 Operationen erlaubt. Für die Konfiguration 5 kommt man dabei auf circa 30.900 Operationen. Dies sind allerdings Annahmen, die auf Schätzungen beruhen.

5.1.4 Testszenario mit 25 Knoten


- Abstand der Pinganfragen: alle 250 Zeitzyklen
- Maximale Paketgröße im Radio: 100 Byte
- Maximale Paketgröße, mit welcher die Aggregations- und Komprimierungsschicht Pakete an die nächste (Richtung Radio) Schicht weiter gibt: 80 Byte

Als Beispielkonfigurationen für die Aggregation wurde sich diesmal auf 6 (Aggregation und Komprimierung) und 9 (reine Aggregation) beschränkt. Beide haben einen Wartezeitraum von 4 Zyklen, nach Ablauf dessen das bis dahin verschmolzene Paket gesendet wird.


In Hinsicht auf das Datenvolumen konnten kaum Verbesserungen erreicht werden. Dieses hielt sich für beide Testszenarien annähernd auf dem Niveau der Referenzmessung (Abbildung 5.14).
5.1.5 Messtabellen

<table>
<thead>
<tr>
<th>Programmkonfiguration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>übergebene Pakete</td>
<td>274</td>
<td>287</td>
<td>291</td>
<td>200</td>
<td>218</td>
<td>302</td>
<td>299</td>
<td>221</td>
<td>329</td>
</tr>
<tr>
<td>gesendete Pakete</td>
<td>274</td>
<td>287</td>
<td>291</td>
<td>119</td>
<td>114</td>
<td>138</td>
<td>226</td>
<td>166</td>
<td>240</td>
</tr>
<tr>
<td>übergebene Byte</td>
<td>5140</td>
<td>5516</td>
<td>5569</td>
<td>3830</td>
<td>4219</td>
<td>5709</td>
<td>5743</td>
<td>4421</td>
<td>6441</td>
</tr>
<tr>
<td>gesendete Byte</td>
<td>5140</td>
<td>5564</td>
<td>4994</td>
<td>3680</td>
<td>3811</td>
<td>5006</td>
<td>6268</td>
<td>4688</td>
<td>6889</td>
</tr>
</tbody>
</table>

Abbildung 5.15.: Komplette Messdaten für 6nodes

<table>
<thead>
<tr>
<th>Programmkonf.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>übergebene Pakete</td>
<td>566</td>
<td>531</td>
<td>496</td>
<td>382</td>
<td>395</td>
<td>507</td>
<td>543</td>
<td>659</td>
<td>752</td>
</tr>
<tr>
<td>gesendete Pakete</td>
<td>566</td>
<td>531</td>
<td>596</td>
<td>217</td>
<td>199</td>
<td>261</td>
<td>431</td>
<td>507</td>
<td>601</td>
</tr>
<tr>
<td>übergebene Byte</td>
<td>11606</td>
<td>10946</td>
<td>10183</td>
<td>7632</td>
<td>7847</td>
<td>10528</td>
<td>11326</td>
<td>13489</td>
<td>15914</td>
</tr>
<tr>
<td>gesendete Byte</td>
<td>11606</td>
<td>11176</td>
<td>9387</td>
<td>7294</td>
<td>7128</td>
<td>9571</td>
<td>12256</td>
<td>14655</td>
<td>17243</td>
</tr>
</tbody>
</table>

Abbildung 5.16.: Komplette Messdaten für line6
5.2 Aufbau und Ablauf der Experimente auf dem RCX

Mit SERNET lassen sich gute Abschätzungen für das Verhalten des Netzverkehrs machen. Laufzeit und Größe des Codes müssen allerdings am praktischen Objekt überprüft werden. Im Rahmen dieser Arbeit wurde dafür die RCX Plattform verwendet.

5.2.1 Betrachtung der Kodegröße

Für die folgenden Werte wurde eine simple „PingPong“-Anwendung zusammen mit einem rudimentären Netzwerkrahmenwerk aus CRC und Radio als Basis verwendet. Zum Vergleich wurde die zu überprüfende Schicht jeweils in unterschiedlichen Konfigurationen in den Netzwerkprotokollstack hinzugefügt.

In Abbildung 5.18 (alle Angaben in Byte) sind die Einträge anhand der Gesamtgröße des kompilierten Kodes geordnet. Die reine Aggregation benötigt knapp 1600 Byte. Je nach freiem Speicherplatz und Bedürfnissen kann die Konfiguration angepasst werden.

<table>
<thead>
<tr>
<th>Programmkonfiguration</th>
<th>Text</th>
<th>Data</th>
<th>BSS</th>
<th>Gesamt</th>
<th>Differenz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keine Aggregation / keine Komprimierung</td>
<td>8384</td>
<td>406</td>
<td>2692</td>
<td>11482</td>
<td>0</td>
</tr>
<tr>
<td>Aggregation ohne Timeout / keine Komprimierung</td>
<td>9822</td>
<td>442</td>
<td>2816</td>
<td>13080</td>
<td>1598</td>
</tr>
<tr>
<td>Aggregation mit Timeout / keine Komprimierung</td>
<td>10300</td>
<td>456</td>
<td>2826</td>
<td>13582</td>
<td>2100</td>
</tr>
<tr>
<td>Keine Aggregation / Komprimierung (RLE&gt;LZW)</td>
<td>10950</td>
<td>454</td>
<td>2704</td>
<td>14108</td>
<td>2626</td>
</tr>
<tr>
<td>Aggregation ohne Timeout / Komprimierung (RLE&gt;LZW)</td>
<td>11716</td>
<td>466</td>
<td>2816</td>
<td>14998</td>
<td>3516</td>
</tr>
<tr>
<td>Aggregation mit Timeout / Komprimierung (BWT&gt;RLE)</td>
<td>11992</td>
<td>480</td>
<td>2830</td>
<td>15302</td>
<td>3820</td>
</tr>
<tr>
<td>Aggregation mit Timeout / Komprimierung (RLE&gt;LZW)</td>
<td>12194</td>
<td>480</td>
<td>2826</td>
<td>15500</td>
<td>4018</td>
</tr>
<tr>
<td>Aggregation mit Timeout / Komprimierung (BWT&gt;MTF&gt;Huffman)</td>
<td>14880</td>
<td>488</td>
<td>2830</td>
<td>18198</td>
<td>6716</td>
</tr>
</tbody>
</table>

Abbildung 5.18.: Messdaten der Kodegröße auf dem RCX

5.2.2 Betrachtung der Laufzeit und des Energieverbrauchs

Als Maß der Laufzeit werden Ticks des Prozessors gezählt, welche direkt aus dem Zählerregister gelesen werden. Abgelesen wurde jeweils der Durchschnittswert nach 400 Paketen, die von der Anwendung übergeben wurden.

Ein weiteres wichtiges Maß ist die Anzahl der Pakete, die im Durchschnitt aggregiert werden konnten, da mit deren Zahl auch die Größe der zu verarbeitenden Daten steigt.

Für die Messung waren vier Knoten gleichzeitig aktiv, welche untereinander Funkkontakt hatten. Die Ergebnisse der Messung sind in Abbildung 5.19 zu finden.

Es ist festzustellen, dass bei allen drei Testkonfigurationen der Timer für einen Flush keine Rolle spielte. Dieser würde erst nach 10 Sekunden, was 20 Paketen entspricht, auslösen.

Überraschend ist der hohe durchschnittliche Wert der Laufzeit für die reine Aggregation ohne Komprimierung. Diese lässt sich teilweise auf den Aufwand für den „Flush“ und den folgenden Neubeginn der Aggregation zurückführen, bleibt aber trotzdem weit über dem Erwarteten. Die Pakete haben eine Größe von 6 Byte und inklusive 1 Byte für die Größeangabe im aggregierten Paket passen in die 45 Byte Grenze 6 Pakete, was die Messung bestätigt.

Die beiden Konfigurationen mit Komprimierung erreichen eine durchschnittliche Aggregation von 13 Paketen. Während die durchschnittliche Laufzeit auf der Empfangsseite annähernd gleich groß ist, braucht die Konfiguration mit BWT für einen Aggregationsdurchlauf mehr Rechenzeit.

Um die Energiebilanz der Konfigurationen zu überprüfen, ist es nötig, dieTicks einer bekannten Größe gleichzusetzen, mit welcher der Energieverbrauch berechnet werden kann. Eine leere For-Schleife mit 1000 Durchläufen benötigt 423 Ticks. Ein Schleifendurchlauf entspricht dabei 30 Prozessorzyklen [12]. Damit erhält man folgende Relation für Ticks zu Prozessorzyklen (Abbildung 5.20):
Eine gesparte Präambel entspricht in etwa 31,73 nicht gesendeten Byte (siehe Kapitel 5.1.3). Für die gemessenen Fälle ergibt das Einsparungen, relativ zu gesendeten Byte, durch die nicht nötigen Präambeln von aggregierten Paketen (Abbildung 5.21):

\[
\text{Präambeleinsparung} \times (\text{aggregierte Pakistan} - 1) = \text{Einsparung je Sendevorgang}
\]

1. \(31,73 \text{ Byte} \times 5 = 158,65 \text{ Byte}\)
2. \(31,73 \text{ Byte} \times 12 = 380,76 \text{ Byte}\)
3. \(31,73 \text{ Byte} \times 12 = 380,76 \text{ Byte}\)

Abbildung 5.21.: Anhand der Aggregation gesparte Byte je Sendevorgang

Kam die Komprimierung zum Einsatz, so konnten 13 Pakete mit jeweils 6 Byte in einem Rahmen von 45 Byte untergebracht werden. Dies entspricht einem Gewinn von 33 Byte je Sendevorgang (Abbildung 5.22).

1. 158,65 Byte
2. 380,76 Byte + 33 Byte = 413,76 Byte
3. 380,76 Byte + 33 Byte = 413,76 Byte

Abbildung 5.22.: Durch Komprimierung und Aggregation gesparte Byte je Sendevorgang

Der genutzte Controller H8/300L benötigt im aktiven Modus bei 16 MHz eine Leistung von 100 mW [13]. Die Arbeitsleistung lässt sich damit anhand der benötigtenTicks berechnen (Abbildung 5.23).

\[
\begin{align*}
\text{n : Anzahl der Ticks} \\
\text{t : reelle Laufzeit s} \\
\text{e : Arbeitsleistung mWs} \\
\text{n} \times 70,92 / 16.000.000 & = t \\
\text{t} \times 100 mW & = e
\end{align*}
\]

Abbildung 5.23.: Formel der Arbeitsleistung des Controllers in Abhängigkeit zu Ticks

\[
\begin{align*}
n : & \text{ Anzahl der Byte} \\
t : & \text{Übertragungszeit} \\
e_s : & \text{Arbeitsleistung auf Sendeseite} \text{mWs} \\
e_r : & \text{Arbeitsleistung auf Empfängerseite} \text{mWs} \\
n \times 0.52 \text{ms} \times 1000 = & t \\
t \times 125 \text{mW} = & e_s \\
t \times 105 \text{mW} = & e_r
\end{align*}
\]

*Abbildung 5.24.: Formel der Arbeitsleistung des Funkmoduls in Abhängigkeit zu den gesendeten Byte*

Aus diesen Zusammenhängen kann nun anhand der gemessenen Werte die Arbeitsleistung und Einsparung berechnet werden (Abbildung 5.25).

<table>
<thead>
<tr>
<th>Alle Werte für Sammel-Pakete</th>
<th>1.) Aggr.</th>
<th>2.) Aggr. / RLE&gt;LZW</th>
<th>3.) Aggr. / BWT&gt;RLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zusätzliche Arbeitsleistung im Controller auf Senderseite</td>
<td>6,92 mWs</td>
<td>12,76 mWs</td>
<td>14,59 mWs</td>
</tr>
<tr>
<td>Zusätzliche Arbeitsleistung im Controller auf Empfängerseite</td>
<td>0,15 mWs</td>
<td>0,79 mWs</td>
<td>0,78 mWs</td>
</tr>
<tr>
<td>Gesparte Arbeitsleistung im Funkmodul auf Senderseite</td>
<td>10,31 mWs</td>
<td>26,89 mWs</td>
<td>26,89 mWs</td>
</tr>
<tr>
<td>Gesparte Arbeitsleistung im Funkmodul auf Empfängerseite</td>
<td>8,66 mWs</td>
<td>22,59 mWs</td>
<td>22,59 mWs</td>
</tr>
<tr>
<td>Durchschnittlich aggregierte Pakete</td>
<td>6</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

*Abbildung 5.25.: Zusätzliche und gesparte Arbeitsleistungen des Testaufbaus*

Geht man theoretisch von stets einem Sender und zwei Empfängern aus, so braucht ein Einzelpaket im Normalfall 6,57 mWs Energie, um gesendet und empfangen zu werden. Für die getesteten Konfigurationen der Aggregation ergeben sich für Einzelpakete folgende Energieeinsparungen (Abbildung 5.26):

- 83 -

5.3 „Burst“-Verhalten der Aggregation

Neben den erwarteten Energieeinsparungen beim Senden von Paketen bringt die Aggregation weitere Vorteile mit sich. Durch verschmolzene Pakete kann der Datendurchsatz positiv beeinflusst werden. Wo sonst mehrere Pakete um die Senderahmen auf MAC-Ebene konkurrieren, teilen sie sich einen einzelnen. Damit können die Daten umgehend an die Nachbarn verteilt werden und die kommenden Senderahmen können für andere Sendevorgänge genutzt werden.


<table>
<thead>
<tr>
<th>1.) Aggr.</th>
<th>2.) Aggr. / RLE&gt;LZW</th>
<th>3.) Aggr. / BWT&gt;RLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Durchschnittlich aggregierte Pakete</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>Durchschnittliche Energieeinsparung pro Paket</td>
<td>3,4 mWs</td>
<td>4,44 mWs</td>
</tr>
<tr>
<td>Durchschnittliche Energieeinsparung pro Paket in Prozent zum Normalfall</td>
<td>52%</td>
<td>68%</td>
</tr>
</tbody>
</table>

Abbildung 5.26.: Durchschnittliche Energieeinsparung pro Einzelpaket
Dieses Verhalten setzt sich fort, wenn die Pakete sich weiter einen Pfad in Richtung Ziel teilen. Bei richtiger Konfiguration wird das verschmolzene Paket entpackt, von den übergeordneten Schichten (zum Beispiel Routing) bearbeitet und ist dann umgehend bereit zum Senden. Mögliche Kollisionen und Konkurrenz um Senderahmen der im aggregierten Paket enthaltenen Teilpakete entfallen so für den kompletten Pfad (Abbildung 5.28).

**5.4 Fazit der Bewertungen**

Mit Hilfe der Aggregation kann die Leistungsfähigkeit des Nachrichtentransports in einem stark frequentieren Netz erhöht werden. Führen zahlreiche Routen über

Die Compression-Klasse mit Aggregation und Komprimierung versagt oder wirkt sogar negativ wenn:

- die logischen Pakete so groß sind, dass zu selten mindestens 2 verschmolzen werden,
- zu wenig Pakete zeitlich nah genug von einem Knoten gesendet werden sollen,
- ein gewählter ergänzender Komprimierungsalgorithmus unangemessen viel Rechenzeit in Anspruch nimmt, was zu zeitlichen Verzögerungen (Zeitspanne des Aggregationsttimeout wird nicht ausgenutzt) und hohem Stromverbrauch führen kann.

Die größeren Pakete führen statistisch auch zu einer höheren Fehlerrate pro Paket. Wie sich dieser Umstand im Zusammenspiel mit der Paketreduktion verhält, kann Inhalt späterer Untersuchungen an praktischen Netzen sein.
6 Ausblick

Der Entwurf bietet einige Möglichkeiten, Verbesserungen einfließen zu lassen. Zum Beispiel global bekannte Verzeichnisse oder Kodierungsbäume ermöglichen spezialisierte Lösungen in der Komprimierung.

Zu Kreuzabhängigkeiten kann es allerdings kommen, wenn andere Schichten Timeouts nutzen und die Compression-Schicht ebenfalls mit Timer-Abhängigkeit konfiguriert wurde. Dann kann es dazu kommen, dass durch die zeitliche Verzögerung in der Aggregation Timeouts anderer Schicht greifen.


Aggregation gepaart mit Komprimierung wird in Funknetzen sicherlich weiter ein Thema sein. Nicht nur in Hinsicht auf mögliche Energieersparnis, sondern sicherlich viel zentraler als Mittel einer möglichen Steigerung des Netzwerkdurchsatzes.
Quellenverzeichnis


[11] Low Power Radio Solutiions Ltd, ERx00-02 Series Data Sheet (Rev 2.3), 2005

[12] Hitachi, H8/300 Programming Manual,