Brandenburgische Technische Universität Cottbus
Institut für Informatik
Lehrstuhl Verteilte Systeme/Betriebssysteme

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

1 Einleitung

Netze aus Funkrobotern, die Sensordaten aufnehmen können, bieten zahlreiche Einsatzmöglichkeiten. Denkbar ist es, zum Beispiel mit einem solchen Sensornetz ein Waldareal auf Brände zu überwachen. Ein Sensorknoten misst dafür aussagekräftige Werte wie zum Beispiel die Temperatur und den Sauerstoffgehalt. Um aus den Werten aller Einheiten einen Überblick über das Areal zu bekommen, müssen diese Daten allerdings zu einer Datensenke geleitet werden. Ein solcher Funkroboter soll jedoch möglichst klein und energiesparend sein, was keine großen Funkreichweiten erlaubt. Sensorknoten können daher oft nur mit wenigen Nachbarn direkten Funkkontakt aufnehmen. Aus diesem Grunde können die Datenpakete nicht immer direkt zur Datensenke gefunkt werden, sondern müssen über andere Knoten geleitet werden. Daraus ergibt sich die Notwendigkeit eines zumindest einfachen Routings. Ist dieses nicht statisch, führt das zu zusätzlichen Daten, die zwischen den Knoten ausgetauscht werden müssen.

Pakete mit Sensordaten und Routinginformationen fließen dabei immer konzentrierter auf die Datensenke zu. Durch die direkten Nachbarknoten der Datensenke werden alle Sensordatenpakete des gesamten Netzes geleitet. Dies führt bei hoher Paketlast zu einem Flaschenhals im Datenverkehr. Allerdings offenbart sich auch ohne hohe Last auf Dauer der Mehrverbrauch von Energie auf diesen Knoten.

Ziel ist es, dass ein solches Sensornetz möglichst lange ohne menschliches Zutun in Betrieb bleibt. Gerade die Einsatzdauer, abhängig vom Energievorrat und -verbrauch eines Sensorknotens, ist ein entscheidender Faktor für die Effizienz eines Sensornetzes. Batteriewechsel oder eine externe Stromversorgung sind in solchen Szenarien nicht praktikabel.

Arbeitet das Funkmodul nur mit einem Kanal, verhält sich das Funkmedium wie ein Bus. Es kann jeweils nur ein Knoten senden, alle Knoten in Reichweite können ihn empfangen. Überträgt man diese Erkenntnis auf einen oben beschriebenen Flaschenhals in der Netztopologie, so muss jedes eingehende Paket, das an die Datensenke adressiert ist, auch wieder gesendet werden. N eingehende Pakete führen zu N ausgehenden Paketen mit dem gleichen Ziel. Wird eine MAC-Schicht benutzt, belegt jedes Paket einen Sende-/Empfangsrahmen, auch wenn es diesen nicht voll ausfüllt.

Abhilfe bringt an dieser Stelle eine Aggregation der Pakete. Alle Pakete haben den gleichen Nachbarknoten als Empfänger, womit es sich offensichtlich anbietet, diese zu einem Paket zu bündeln und dieses auf der Empfängerseite wieder in seine Teilpakete

zu zerlegen. Der Vorteil des Funks ist, wie oben erwähnt, dass jeder direkte Nachbar den Sendeknoten hört. Daher kann man die Aggregation auf alle ausgehenden Pakete anwenden.
Durch die Aggregation wachsen die sonst eher kleinen Pakete von Sensornetzen unter Umständen beträchtlich im Datenvolumen. Unter dem Ziel der Energieersparnis ist es in Betracht zu ziehen, ob eine Komprimierung dieser Daten zu einer längeren Laufzeit der Sensorknoten beitragen kann.
Da jedes Byte, das man über Funk sendet, einen gewissen Energieaufwand verlangt, kann hier die Lebensdauer einer Sensoreinheit positiv beeinflusst werden, solange der Rechenaufwand für das Komprimieren klein genug bleibt. Laut Abschätzungen kann dabei von etwa 1.000 Byte-Operationen zu 1 übertragenen Byte ausgegangen werden [1].
Bei der späteren Implementierung wird zudem auf die spezielle Umgebung mit ihren Limitierungen zu achten sein. Die Leistung der Prozessoren ist nicht vergleichbar mit der von Desktopsystemen und auch der Arbeitsspeicher ist ein weit wertvolleres Gut.

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

Um die Möglichkeit und die Effizienz der Komprimierung zu bewerten, braucht es einige essentielle Maße und Begriffsklärungen. In der geplanten Umsetzung wird jedes Paket/ jede Nachricht separat betrachtet, ohne auf eine gemeinsame Quelle zu schließen. Daher sind im Folgenden die Begriffe Quelle und Nachricht als gleich zu betrachten.
Eine Nachricht besteht aus der Auswahl einer definierten Menge von Symbolen/Zeichen, beispielsweise Buchstaben.
Eine Quelle hat eine bestimmte Anzahl n von verschiedenen Symbolen, welche als Symbolvorrat bezeichnet wird (Abbildung 2.1).

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

Abbildung 2.1.: Symbolvorrat

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).

$$n : Anzahl\ Symbole\ der\ Nachricht$$
$$H_0 = m = ld(n)$$

Abbildung 2.2.: Formel des Entscheidungsgehalt H_o

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).

n : Anzahl Symbole der Nachricht
n_i : Anzahl Symbol i der Nachricht

$$p_i = \frac{n_i}{n}$$

$$\sum_{i=1}^{n} p_i = 1$$

Abbildung 2.3.: Formel der Auftrittswahrscheinlichkeit p_i

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.

n : Anzahl Symbole der Nachricht
p_i : Wahrscheinlichkeit des Symbols i

$$H = -\sum_{i=1}^{n} p_i * ld\left(\frac{1}{p_i}\right)$$

Abbildung 2.4.: Formel der Entropie H

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).

$$H \leq H_0$$

Abbildung 2.5.: Regel der Maximalentropie

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).

H_0 : Entscheidungsgehalt
H : Entropie

$$R = H_0 - H$$

Abbildung 2.6.: Formel der Redundanz

Die mittlere Kodewortlänge Hc einer Nachricht berechnet sich ähnlich wie die Entropie,

nur dass man für jedes Symbol die Bitlänge der Symbolkodierung direkt einsetzt. Hc ist die entscheidende Größe für die Einschätzung von Entropiekodierungen (Abbildung 2.7).

n : *Anzahl Symbole der Nachricht*
p_i : *Wahrscheinlichkeit des Symbols i*
m_i : *Kodelänge des Symbols i*

$$H_c = -\sum_{i=1}^{n} p_i * m_i$$

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. Entropiekodierern müssen zu Beginn die Verteilungshäufigkeiten bekannt sein oder diese werden erst im Laufe der Verarbeitung dynamisch ermittelt, um ein Zeichen anhand seiner Kodewortlänge zu kodieren. 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: AAAAAAAAAABBBBBCCCCDDAAAAA

Abbildung 2.8.: Beispiel Wiederholung von Einzelsymbolen

Musterwiederholung findet man zum Beispiel häufig in Texten. Damit sind Wiederholungen von Symbolgruppen gemeint. Im Beispiel in der Abbildung 2.9 wäre dies die Zeichenkombination “BC” oder auch “ABC”.

Beispiel: ABCRTHABCNJEABCHRWTBCGESBCHDBC

Abbildung 2.9.: Beispiel Musterwiederholung

Als **Ortsredundanzen** bezeichnet man Redundanzen, die nach einem bestimmten Muster im Datenstrom auftreten. Zum Beispiel eine Linie oder ein Kreis in einem Bild. Beispiel: Das A in der Abbildung 2.10 tritt alle 3 Symbole auf und das B vor allem am Nachrichtenanfang.

Beispiel: ABCABBADBABFAGHAKRANEAJE

Abbildung 2.10.: Beispiel Musterwiederholung

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.

Beispiel: ABACABACABACABACABAC

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.

```
AAAACBBBCDEAAAAA
```

Abbildung 2.12.: Eingangsnachricht für die Lauflängenkodierung

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.

```
4*A1*C3*B1*C1*D1*E5*A
```

Abbildung 2.13.: Denkbare Kodierung der Nachricht durch Lauflängenkodierung

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

```
4A1C3B1C1D1E5A
```

Abbildung 2.14.: Überdachte Kodierung der Nachricht durch Lauflängenkodierung

Im Vergleich zu den Eingangsdaten hat sich die Zeichenmenge von 16 auf 14 verringert. Das ist bereits ein beachtlicher Erfolg. Das C wird jedoch mit zwei Zeichen dargestellt, obwohl es in den Ursprungsdaten nur ein Zeichen einnahm. Einzelne Zeichen kommen in normalen Datenmengen sehr häufig vor und würden dabei jedes mal das Doppelte an Speicherplatz in Anspruch nehmen. Dieser Effekt ist alles andere als wünschenswert und würde den Algorithmus nur in speziellen Fällen zum Erfolg verhelfen.
Um solche einzelne Zeichen weiterhin mit dem Platzverbrauch von 1 zu repräsentieren, ist ein sogenanntes Escape-Zeichen notwendig, welches jede RLE-Kodierung einleitet. Dadurch bleibt der Rest der Eingabezeichenkette unangetastet. Nachteil ist jedoch, dass jede RLE-Kodierung nun 3 Zeichen („Escape-Zeichen – Anzahl – Zeichen“) statt wie vorher 2 („Anzahl – Zeichen“) benötigt. Obiges Beispiel sieht bei der Benutzung von “$” als Escape-Zeichen wie in Abbildung 2.15 aus.

$4AC$3BCDE$5A

Abbildung 2.15.: Kodierung der Nachricht durch Lauflängenkodierung, wie sie in der Praxis angewandt wird

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

Problematisch wird allerdings die Kodierung des als Escape-Zeichen reservierten Zeichens in der Eingangszeichenkette. Zahlreiche Implementierungen sehen hier eine Ausweichstrategie vor, in der das Escape-Zeichen über das Standardkodierungstupel dargestellt wird, also Escape-Zeichen, Anzahl(1), Escape-Zeichen. Tritt das Escape-Zeichen einzeln auf, ist das eine Datenerhöhung von 1 auf 3. Diesem Effekt kann wie folgt entgegengewirkt werden.

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 Zweitertupel „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.

Der Anzahlwert hat damit immer noch die ungenutzten Werte 1, 2 und 3. Solange man diesen keine Sonderfunktion zuerkennt, ist es vorausschauend, den eigentlichen Wert bei der Kodierung um 3 zu dekrementieren und bei der Dekodierung diese Änderung wieder rückgängig zu machen. Auf diese Weise können längere Zeichenfolgen in ein Dreiertupel verpackt werden. Normalerweise müsste nach 255 (bei Kodierung in Bytes) gleichen Zeichen ein neues Tupel beginnen, um einen Überlauf zu verhindern. Durch das Inkrementieren lassen sich so aber Folgen mit bis zu 258 Zeichen auf einmal kodieren.

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.
Ein Verfahren wäre zum Beispiel, das Escape-Zeichen als Bit darzustellen. Wird von einer Bytekodierung ausgegangen und das erste Bit jedes Zeichens übernimmt die Aufgabe des Escape-Zeichens, so sind nur 128 Zeichen normal mit einem Zeichen zu kodieren. Alle anderen Zeichen, die durch das sinnentfremdete Bit nicht direkt speicherbar sind, müssten dann ähnlich der Escape-Zeichenkodierung im normalen Algorithmus gesondert kodiert werden.
Eine im Ansatz ähnliche Modifizierung des RLE-Algorithmus' verwendet das Bildformat PCX der Firma Zsoft [3]. Nur Zeichen mit Werten bis 192 werden direkt abgebildet. Kommt es zu einer Wiederholung wird die Anzahl mit 192 addiert und als Zweitertupel mit dem Zeichenwert zusammen gespeichert. Der Dekoder erkennt Werte größer als 192 als Beginn einer RLE-Kodierung. Eindeutiger Vorteil ist die Länge der Kodierung, welche 2 statt 3 Stellen lang ist. Dies muss jedoch abgewogen werden mit dem Umstand, dass Zeichenwerte über 192 ähnlich dem Escape-Zeichen gesondert kodiert werden müssen und Wiederholungen nur bis zu einer Länge von 64 Stellen in einem Tupel kodiert werden können. Die Grenze von 192 kann allerdings der Problemstellung angepasst werden.
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:
Die Lauflängenkodierung verbraucht in der Ausführung praktisch keinen zusätzlichen Speicher und ist extrem schnell. Allerdings kommt sie nur mit Wiederholungen von einzelnen Symbolen klar (Versionen, die mehrere Symbole erkennen, sind Spezialfälle der normalen Lauflängenkodierung). Sie führt zwar nicht immer zum Erfolg, vergrößert die Ausgabe aber nur in seltenen Fällen, in denen viele Escape-Zeichen in den Eingangsdaten vorhanden sind.

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

Die Wortkodierung ist auch unter der Bezeichnung Token-basierte Komprimierung bekannt. Dabei wird ein Dokument als Ansammlung aus vorkommenden Wörtern/Zeichenketten betrachtet. Diese werden in einem Verzeichnis (Dictionary) eingetragen und der Index des Eintrags ersetzt die Zeichenkette im Dokument.
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.

Der Grundgedanke der Wortkodierung wird mit den Algorithmen der LZ-Linie aufgegriffen und verfeinert. Ausgangspunkt ist der LZ77 Algorithmus, welcher mehrfach optimiert und überarbeitet wurde. Teilweise bekamen recht einfache Optimierungen gleich einen eigenen “LZ”-Namen. Um jedoch nicht den Überblick zu verlieren, werden solche Minimaloptimierungen im Folgenden nicht als separater Algorithmus aufgeführt, sondern finden nur im Text Erwähnung. Aufgrund der Fülle von LZ-Derivaten werden einige auch keine Erwähnung finden. Nur als Veranschaulichung hier eine unvollständige Liste von Derivaten, die bei der Recherche aufgefallen sind: LZ77, LZ78, LZAP, LZB, LZC, LZFG, LZH, LZJ, LZMW, LZO, LZP, LZPP, LZR, LZRW1, LZSS, LZT, LZV, LZW, LZY.

2.3.2.1 LZ77

1977 stellten Jacob Ziv und Abraham Lempel erstmals ein Verfahren zur Komprimierung vor, das sich effizient mit der Wortkodierung befasst. Es handelt sich im Grunde um eine Wortkodierung, nur mit der Besonderheit, dass kein Verzeichnis aufgebaut wird, sondern der Teil der Nachricht, welcher bereits kodiert wurde, als eine Art Verzeichnis genutzt wird.

Es kommt ein “Sliding-Window”-Verfahren zum Einsatz. Der Algorithmus bewegt sich bei der Kodierung mit einem sogenannten Fenster über die zu kodierende Nachricht. Ein Teil des Fensters betrachtet über eine gewisse Zeichenlänge noch zu kodierende Symbole, der andere Teil behält bereits kodierte Symbole der Eingangsnachricht als Verzeichnis im Fokus.

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.

Suchfenster								Vorschau				Ausgabe		
0	1	2	3	4	5	6	7	0	1	2	3	P	L	S
								A	N	A	N	0	0	A

Abbildung 2.19.: Kodierungsschritt 1 LZ77

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.

Suchfenster								Vorschau				Ausgabe		
0	1	2	3	4	5	6	7	0	1	2	3	P	L	S
							A	N	A	N	A	0	0	N

Abbildung 2.20.: Kodierungsschritt 2 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.

Suchfenster								Vorschau				Ausgabe		
0	1	2	3	4	5	6	7	0	1	2	3	P	L	S
						A	N	A	N	A	S	6	3	S

Abbildung 2.21.: Kodierungsschritt 3 LZ77

Das Suchfenster, welches als Verzeichnis dient, beinhaltet nun die Phrase „AN“. Diese ist auch in der Vorschau enthalten (Abbildung 2.21). Nun könnte man bereits als Ausgabe das Tripel 6,2,A schreiben. Allerdings lässt sich in Fällen wie diesem weiter optimieren. Die gefundene Phrase hat eine Symbollänge von 2. Damit kennt der Dekodierer später auch die ersten 2 Symbole in der Vorschau des Kodierers. Die gefundene Phrase im Suchfenster endet an der Stelle, an welcher die Vorschau beginnt. Da deren Anfang auch dem Dekodierer bekannt ist, kann man sie in die Kodierung mit einbinden. In unserem Beispiel kann dies genutzt werden, da hierdurch die Phrase „ANAS“ statt nur „ANA“ kodiert werden kann. Somit wird ein Kodierungsschritt und

damit ein Tripel gespart.
Die Ausgabe sieht dann wie folgt aus (Abbildung 2.22):

0	0	A	0	0	N	6	3	S

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.

	0	0	A	0	0	N	6	3	S
Darstellungs-länge	3	2	8	3	2	8	3	2	8

Abbildung 2.23.: Datengröße der kodierter Nachricht des LZ77 Algorithmus

Das Suchfenster im Beispiel hat eine Länge von 8 und kann so mit 3 Bit dargestellt werden. Die Vorschau betrachtet 4 Symbole, wodurch die Phrasenlänge durch 2 Bit beschrieben werden kann. Das folgende Symbol bleibt in 8 Bitdarstellung. Es ergibt sich dadurch eine Ausgangsnachrichtenlänge von 39 Bit (Abbildung 2.23), was gegenüber der Eingangsnachricht von 48 Bit bereits ein gute Komprimierung darstellt. Diese Optimierung mit variablen Bitlängen für die Position und Phrasenlänge wird auch als LZB [4] bezeichnet.

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

LZSS ist ein verbessertes Verfahren von LZ77, das 1982 von James Storer und Thomas Szymanski [5] vorgestellt wurde.

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.

Eine weitere Ergänzung zu den Ideen von Ziv und Lempel ist die Nutzung eines Ringpuffers für das “Sliding-Window”. Dadurch haben die Daten im Suchfenster für eine bestimmte Zeit eine feste Adresse im Speicher, wodurch sich ein Suchbaum über den Daten aufbauen lässt und der Suchaufwand deutlich sinkt.

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

Suchfenster								Vorschau				Ausgabe		
0	1	2	3	4	5	6	7	0	1	2	3	B		S
								A	N	A	S	0		A
							A	N	A	N	A	0		N

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).

Suchfenster								Vorschau				Ausgabe		
0	1	2	3	4	5	6	7	0	1	2	3	B	P	L
						A	N	A	N	A	S	1	6	3

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).

Suchfenster								Vorschau				Ausgabe		
0	1	2	3	4	5	6	7	0	1	2	3	B		S
			A	N	A	N	A	S				0		S

Abbildung 2.26.: Kodierungsschritt 4 für LZSS

Abschließend noch einmal ein Einzelsymbol angeführt vom Signal-Bit 0 (Abbildung 2.26).

Zählt man alle Bits, die für die Kodierung der Eingangsnachricht nötig sind, zusammen, so erhält man 33 Bit (Abbildung 2.27). Die gleiche Beispielnachricht 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.

	0	A	0	N	1	6	3	0	S
Darstellungslänge	1	8	1	8	1	3	2	1	8

Abbildung 2.27.: Datengröße der kodierter Nachricht des LZSS Algorithmus

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

Ein Jahr nachdem Ziv und Lempel ihre Arbeiten veröffentlichten, die heute als LZ77 bekannt sind, schlugen sie selber einige Verbesserungen am Verfahren vor [6].

Statt dem “Sliding-Window”-Prinzip für das Verzeichnis führten sie nun ein solches in Tabellenform ein. Damit das Verzeichnis nicht mit der kodierten Nachricht mitgeschickt werden muss, wird es dynamisch während des Kodierens und Dekodierens aufgebaut. Bei Schritt n beider Kodierungsrichtungen ist das Verzeichnis auf beiden Seiten exakt das Gleiche. Zu Beginn ist das Verzeichnis leer bis auf den ersten Eintrag, in dem der “Nullstring” steht. Dessen Index hat den Charakter eines Escape-Zeichens.

Beim Komprimieren versucht das Verfahren von der aktuellen Symbolposition an, die längste Zeichenkette im Verzeichnis zu finden. Deren Index wird als Ausgabe geschrieben. Diesem wird ähnlich dem LZ77-Algorithmus noch das Symbol angehängt, welches der Zeichenkette in der Eingangsnachricht folgt. Dies soll wieder anhand der Nachricht „ANANAS“ demonstriert werden.

Index	Eintrag
0	„Nullstring“
Restnachricht	Ausgabe
ANANAS	0,A

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).

Index	Eintrag
0	„Nullstring“
1	A
Restnachricht	Ausgabe
NANAS	0,N

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).

Index	Eintrag
0	„Nullstring“
1	A
2	N
Restnachricht	Ausgabe
ANAS	1,N

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.

Index	Eintrag
0	„Nullstring“
1	A
2	N
3	AN
Restnachricht	Ausgabe
AS	1,S

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:

0	A	0	N	1	N	1	S

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

LZW stellt sicherlich die populärste Modifikation der LZ78-Linie dar. Sie wurde 1984 von Terry A. Welch vorgestellt [7]. Die beiden entscheidenden Unterschiede sind zum einen ein Verzeichnis, das bereits mit allen Zeichen des Alphabets vorinitialisiert ist, und zum anderen wird das Folgesymbol einer kodierten Phrase nicht automatisch an den Index der kodierten Nachricht angehängt. Es gibt also keine Tupel oder gar Tripel, wie bei LZ77 und LZ78, sondern nur den Index, der als kodiertes Symbol eingetragen wird.
Das vorinitialisierte Verzeichnis hat den Vorteil, dass der "Nullstring"-Index als Escape-Zeichen wegfällt, um bisher nicht verwendete Symbole klar zu kodieren. Nun ist es möglich, gleich einen Index in die kodierte Nachricht zu schreiben.
Dass nun nicht mehr immer ein einzelnes Symbol an den geschriebenen Index angehängt wird, erlöst den LZ78-Algorithmus von einem Detailproblem. Folgt auf eine bekannte Phrase, die kodiert werden konnte, wiederum ein bekannte Phrase, so wurde diese bei LZ78 auseinandergerissen. Als Beispiel der Teiltext in der folgenden Abbildung 2.33:

[...]LZ77LZ78[...]

Abbildung 2.33.: Eingangsnachricht für LZW

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

256	LZ7
257	LZ77

Abbildung 2.34.: Startverzeichnis für LZW

Beginnt LZ78 den oben gezeigten Teilabschnitt zu kodieren, findet es mit 257 den längsten passenden Eintrag im Verzeichnis. Diesen schreibt es in die Ausgabenachricht zusammen mit dem Folgesymbol der Nachricht, in diesem Fall L. Nächste Position für den Kodierungalgorithmus wäre dann das Z. Damit beraubt sich der Algorithmus allerdings der Chance, das folgende LZ7 als bekannte Phrase zu erkennen und zu kodieren.
Das Anhängen des Folgesymbols ist jedoch unnötig, da das erste Symbol der folgenden Phrase ja beiden Seiten bekannt ist. Aus diesem Grund fällt dieser Schritt beim LZW-Verfahren weg. Die Bitzahl der Indexangabe wächst entsprechend mit dem Verzeichnis,

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

Fazit: Für LZW gilt Ähnliches wie für LZ78. Das Verzeichnis benötigt zusätzlichen Speicher, beschleunigt dafür die Suche von bekannten Phrasen. Allerdings ist bei LZW in den meisten Fällen mit besseren Ergebnissen zu rechnen. Da die Indexangabe variabel ist, ergibt sich keine weitere Redundanz in der Ausgabe.

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.
Bei der Entropiekodierung werden den Symbolen Codes unterschiedlicher Länge zugewiesen. Häufige Symbole werden zum Beispiel mit weniger Bits dargestellt, als seltene Symbole. Die Kodierung der Zeichen mit einer variablen Anzahl von Bits stellt allerdings ein Problem dar, insbesondere wenn man bedenkt, dass die Entropie für fast alle Wahrscheinlichkeiten eine rationale Zahl ergibt. Doch auch dafür gibt es bereits Verfahren, die in diesem Abschnitt Erwähnung finden werden.
Geht man von dem einfachen Fall aus, dass für Symbole unterschiedlich lange Bitfolgen genommen werden, so stößt man relativ schnell auf eine Einschränkung. Bei der Kodierung muss darauf geachtet werden, dass der empfangene Code eindeutig rekonstruierbar ist. Das heißt, dass keine Mehrdeutigkeiten auftreten.

Beispiel:
Eine uneindeutige Kodierung (Abbildung 2.35)

Symbol	Bitfolge
A	0
B	01
C	11
D	00

Abbildung 2.35.: Nicht eindeutige Symbolkodierung

Nachricht	0	0	1	1	1	1
Beispielinter-	A	A	C		C	
	D		C		C	
pretationen	A	B		C		?

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

Symbol	Bitfolge
A	0
B	10
C	110
D	111

Abbildung 2.37.: Eindeutige Symbolkodierung

kann jedes Symbol zweifelsfrei erkannt werden, sobald sein Kode vollständig vorliegt (Abbildung 2.38).

Code	0	0	1	1	1	1
Nachricht	A	A	D			A

Abbildung 2.38.: Dekodierung einer Nachricht, dessen Kodierung eindeutig ist

2.3.3.1 Shannon-Fano-Kodierung

Um die Symbole einer Nachricht mit verschiedenen Bitlängen zu kodieren, muss es eine Vorschrift geben, die die rationale Auftrittswahrscheinlichkeit in eine ganzzahlige Bitlänge überführt. Claude E. Shannon und Robert Fano stellten 1948 einen Algorithmus vor, der diese Problemstellung löst [2]. Mit seiner Hilfe erhält man einen binären Baum, dessen Blätter die Symbole darstellen und die Kanten bis dorthin den Bitkode. Auf diese Weise bekommt man einen eindeutigen Kode, da jeder Pfad vom Wurzelknoten durch ein Symbol im Blatt abgeschlossen wird.
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).

Symbol	E	S	B	T	I	M	R	Y
n_i	3	3	2	2	1	1	1	1

Abbildung 2.40.: Symbolhäufigkeiten sortiert

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

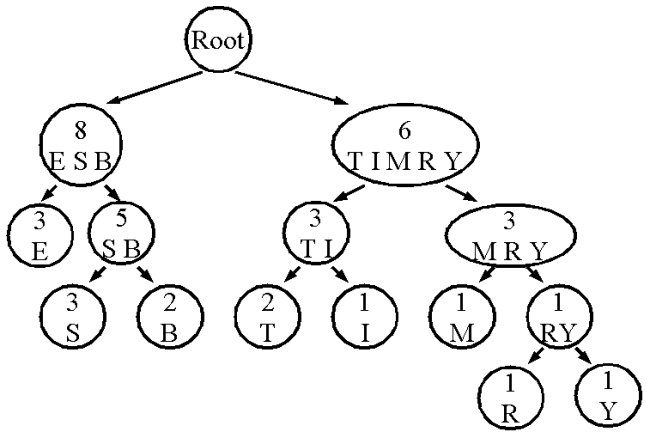

Abbildung 2.41.: Aufbau des Kodierungsbaums nach Shannon-Fano

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).

Symbol	E	S	B	T	I	M	R	Y
Bitlänge	2	3	3	3	3	3	4	4

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):

H_{sf} : mittlere Symbollaenge Shannon − Fano
$H_0 = 3\ Bit/Symbol$
$H = 2.8424\ Bit/Symbol$
$H_{sf} = 2.9286\ Bit/Symbol$
$R = H_0 - H = 0.1576$
$R_{sf} = H_{sf} - H = 0.0862$

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):

Symbol	E	S	B	T	I	M	R	Y
Kode	00	010	011	100	101	110	1110	1111

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

Diese Kodierungen werden einfach hintereinander gereiht anstelle der Symbole der Quellnachricht. Bei der Dekodierung wird ähnlich der Kodierung vorgegangen. Mit dem ersten Bit wird beim Wurzelknoten begonnen und dann anhand der eingelesenen Bits den Kanten des Baumes gefolgt. Wird ein Blatt erreicht, wird dessen Symbol in die Ausgabe geschrieben und es wird wieder beim Wurzelknoten begonnen. Dies wird solange wiederholt, bis die kodierte Nachricht vollständig abgearbeitet wurde.
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:

- Erstelle einen „Wald“ mit Bäumen, für jedes Symbol. Diese Bäume enthalten anfangs nur das Symbol als einen Knoten.
- Suche die beiden Bäume im Wald, die für die Symbole mit der kleinsten Häufigkeit stehen. Entferne diese Bäume aus dem Wald. Erstelle einen neuen Baum, der die beiden entfernten Bäume als Unterbaum hat. Füge diesen Baum in den Wald ein. Benutze dabei die Summe der Häufigkeiten der Unterbäume.
- 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.

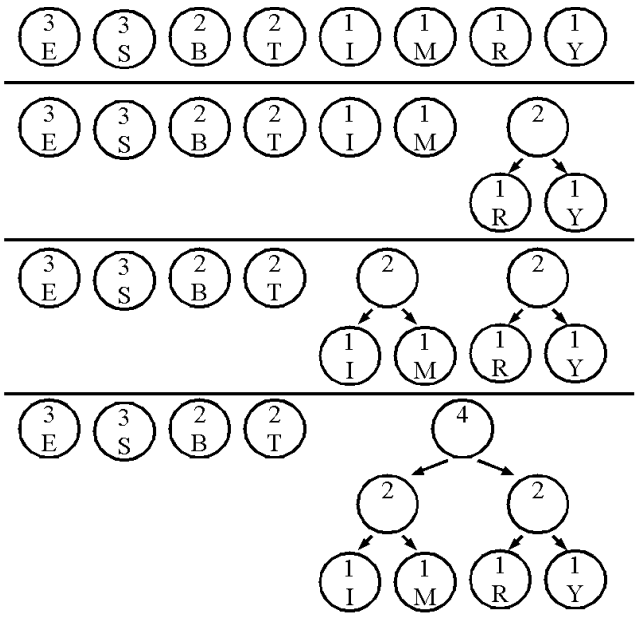

Abbildung 2.45.: Aufbau eines Kodierungsbaums nach Huffman - Teil 1

Wie in der Abbildung 2.45 zu sehen, werden immer Bäume aus dem Wald zusammengefügt, deren Summe aller enthaltenen Symbole am geringsten ist. Bereits hier deutet sich an, dass der Kodierungsbaum am Ende einen anderen Aufbau haben wird, als der nach Shannon-Fano. Weitere Schritte sind in der Abbildung 2.46 dargestellt.

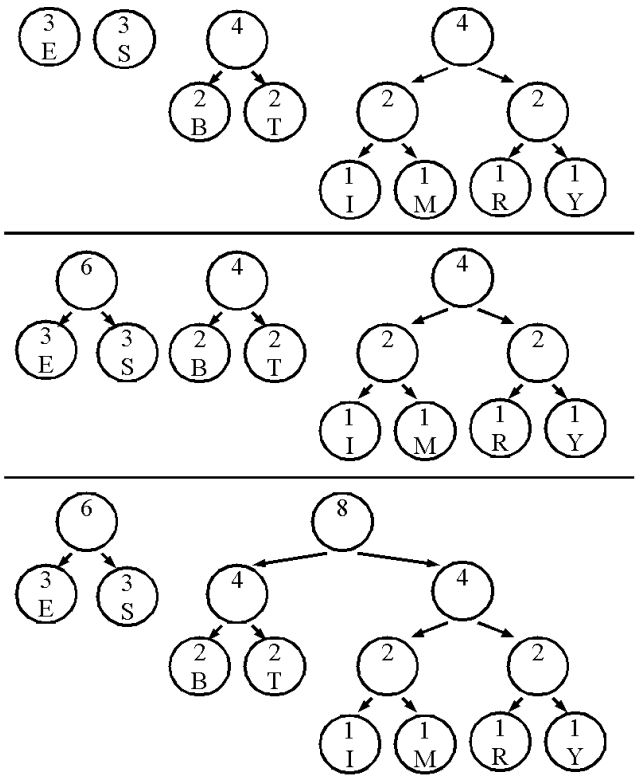

Abbildung 2.46.: Aufbau eines Kodierungsbaums nach Huffman – Teil 2

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.

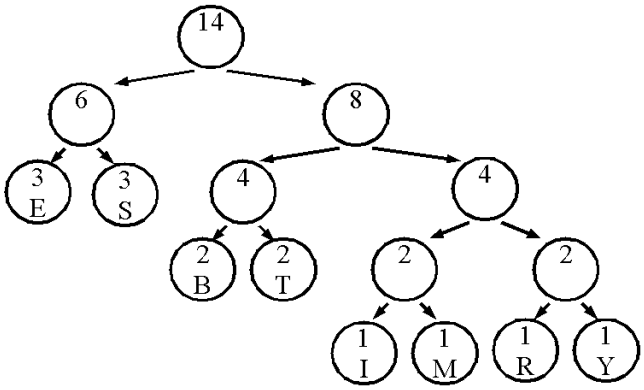

Abbildung 2.47.: Aufbau eines Kodierungsbaums nach Huffman – Teil 3

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

H_h : *mittlere Symbollaenge Huffman*
$H_0 = 3\ Bit/Symbol$
$H = 2.8424\ Bit/Symbol$
$H_{sf} = 2.9286\ Bit/Symbol$
$H_h = 2.8571\ Bit/Symbol$
$R = H_0 - H = 0.1576$
$R_{sf} = H_{sf} - H = 0.0862$
$R_h = H_h - H = 0.0148$

Abbildung 2.48.: Vergleichsmaße der Kodierung aus dem Huffman-Algorithmus

Betrachtet man den Wert der Redundanz in Abbildung 2.48 ist deutlich eine Verbesserung durch die Huffman-Kodierung eingetreten. Diese Verbesserung ist nicht immer zwingend, da ein Shannon-Fano-Kodierungsbaum auch einen optimalen Kodierungsbaum darstellen kann.

Fazit:
Der Speicheraufwand ist in der Zeit, in der der Huffman-Baum aufgebaut wird, am höchsten. Danach muss lediglich der Bitkode für die Symbole im Speicher gehalten werden. Bei der Baumerstellung ist auch der Rechenaufwand am höchsten, da hier mehrmals kurze Sortierungen vorgenommen werden müssen. Danach ist aber der Rechenaufwand gering, da es lediglich eine Übersetzung der Eingangssymbole in den Bitkode gibt.
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?
Erste und naheliegendste Lösung ist das Mitschicken mit der kodierten Nachricht. Ist ein binärer Baum nach der Tiefe seiner Blätter sortiert, so lässt sich exakt der gleiche Baum allein aus dem Wissen der Reihenfolge der Blätter und ihrer Tiefe wiederherstellen. Natürlich muss auf der Senderseite bereits mit dem sortierten Baum kodiert worden sein, damit die Dekodierung funktioniert. Nehmen wir als Beispiel den Kodierungsbaum aus dem Abschnitt Huffman-Kodierung. Dieser ist bereits von links nach der Tiefe seiner Blätter sortiert, wodurch dieser Schritt entfällt.
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:

0	2	E	S	2	B	T	4	I	M	R	Y

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.
Eine weitere Möglichkeit wäre es, den Baum auf der Empfängerseite dynamisch anhand von Informationen der Senderseite aufzubauen. Dies erfordert allerdings viel Arbeit mit Baumstrukturen, welche ständig aktualisiert und neu aufgebaut werden müssten. Für das Ziel der tief eingebetteten Systeme und die Aufgabe des Komprimierens für Energieeinsparungen ist das allerdings ein sehr ungünstiger Umstand, der diese Option praktisch ausschließt.

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 Kodewortelänge 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

letzten Teilintervall die Nachricht komplett dekodiert werden. Dabei wird idealerweise ein Wert gewählt, der möglichst wenig Nachkommastellen aufweist und somit am wenigsten Bits zur Speicherung benötigt. Verständlicherweise muss der Dekodierer dafür die Intervalleinteilung der Kodiererseite kennen.

Der Dekodierung dient das Startintervall mit der gleichen Einteilung wie beim ersten Kodierungsschritt als Ausgang. Das Teilintervall, in dem der übermittelte Wert liegt, steht für das kodierte Zeichen. Auch hier dient dieses Intervall als Grundlage für den nächsten Schritt.

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.

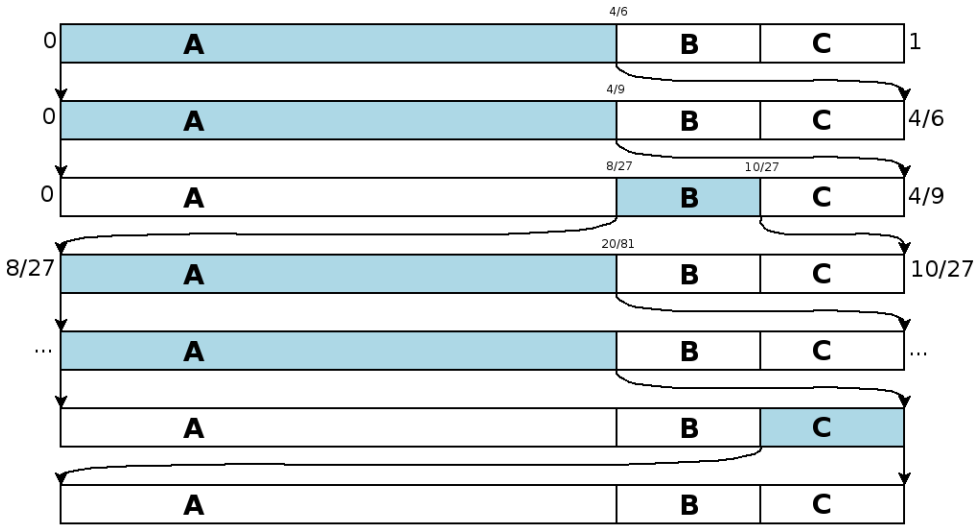

Abbildung 2.50.: schematische Teilintervalle der Arithmetischen Kodierung

0	0.666666667	0.833333333	1 A
0	0.444444444	0.555555556	0.666666667 A
0	0.296296296	0.37037037	0.444444444 B
0.296296296	0.345679012	0.358024691	0.37037037 A
0.296296296	0.329218107	0.33744856	0.345679012 A
0.296296296	0.31824417	0.323731139	0.329218107 C
0.323731139			0.329218107

Abbildung 2.51.: numerische Teilintervalle der Arithmetischen Kodierung

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.

Fazit: Die Arithmetische Kodierung ist auf tief eingebetteten Systemen problematisch, da sie sehr auf Gleitkommazahlen angewiesen ist. Alternativen mit Ganzzahlen benötigen große Datentypen, die nativ von kaum einem eingebetteten System zur Verfügung gestellt werden.

2.3.4 Unterstützende Verfahren

Neben den komprimierenden Verfahren gibt es noch jene, welche die Eingangsdaten vorbereiten und besser komprimierbar machen. Dabei werden die Daten bei einigen sogar vorübergehend größer.
Die meisten dieser Verfahren wandeln dabei eine Art der Redundanz in eine andere, leichter komprimierbare Redundanz um.

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“. M. Burrows und D. Wheeler haben ein Verfahren vorgestellt [9], das diesen Zusammenhang reversibel und platzsparend in die einfacher zu komprimierende Symbolwiederholung (siehe: Lauflängenkodierung Kapitel 2.3.1) und Ortsredundanzen (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:

.ANANAS.

Abbildung 2.52.: Eingangsdaten für Burrows-Wheeler-Transformation

Wie man bei genauerer Betrachtung erkennen kann, ist vor einem „N“ immer ein „A“ und vor einem „A“ fast immer ein „N“. Diese einfache Musterwiederholung sollte sich im Ergebnis der Transformation als Häufung von Symbolwiederholungen niederschlagen.

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

.	A	N	A	N	A	S	.
.	.	A	N	A	N	A	S
S	.	.	A	N	A	N	A
A	S	.	.	A	N	A	N
N	A	S	.	.	A	N	A
A	N	A	S	.	.	A	N
N	A	N	A	S	.	.	A
A	N	A	N	A	S	.	.

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).

A	N	A	N	A	S	.	.
A	N	A	S	.	.	A	N
A	S	.	.	A	N	A	N
N	A	N	A	S	.	.	A
N	A	S	.	.	A	N	A
S	.	.	A	N	A	N	A
.	A	N	A	N	A	S	.
.	.	A	N	A	N	A	S

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.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 Ursprungsnachricht 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

				.	A	N	A
				N	A	N	A
				N	A	S	.
				A	N	A	N
				A	N	A	S
				A	S	.	.
				.	.	A	N
				S	.	.	A

Abbildung 2.58.: Rücktransformation nach Burrows-Wheeler im dritten 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).

A	N	A	N	A	S	.	.
A	N	A	S	.	.	A	N
A	S	.	.	A	N	A	N
N	A	N	A	S	.	.	A
N	A	S	.	.	A	N	A
S	.	.	A	N	A	N	A
.	A	N	A	N	A	S	.
.	.	A	N	A	N	A	S

Abbildung 2.59.: Rücktransformation nach Burrows-Wheeler im letzen Schritt

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

Fazit: BWT wandelt Musterwiederholungen gut in Wiederholungen von Zeichen um. Dabei muss zusätzlich nur ein Indexwert übertragen werden. Ins Gewicht fällt dagegen der Rechenaufwand der Sortierungen, der mit der Nachrichtengröße deutlich wächst. BWT lässt sich gut als Vorstufe für eine Lauflängenkodierung oder die folgende MTF-Transformation nutzen. Die Speichernutzung ist in der Praxis nicht quadratisch zur Nachrichtengröße, sondern kann linear umgesetzt werden (siehe Implementierung von `BwtCodec`).

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 Beispieleingabe 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).

Eingabedaten	Alphabet	Ausgabedaten
.NNAAA.S	ANS.	4
.**N**NAAA.S	.ANS	43
.N**N**AAA.S	N.AS	431
.NN**A**AA.S	N.AS	4313
.NNA**A**A.S	AN.S	43131
.NNAA**A**.S	AN.S	431311
.NNAAA**.**S	AN.S	4313113
.NNAAA.**S**	.ANS	43131134

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

Hatte man vor der Transformation ein Verhältnis der Symbole im Alphabet von „ANS.“=3:2:1:2 , so stellt sich dieses nach der Transformation günstiger als „134“=3:3:2 dar.

H : Entropie vor MoveToFront
H_c : Entropie nach MoveToFront
$H = 1{,}9056\ Bit/Symbol$
$H_c = 1{,}5613\ Bit/Symbol$

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.
Fazit: Die Move-To-Front-Transformation ist eine sinnvolle Ergänzung als Vorstufe von Entropiekodierern. Die Anzahl der Zeichen bleibt gleich, da keine Zusatzinformationen übertragen werden müssen. Der Rechenaufwand ist gering.

2.4 Kombination der Komprimierungsalgorithmen

Viele Komprimierungsalgorithmen behandeln nur eine Redundanzart oder überführen diese in eine andere. Erst in Kombination spielen viele Verfahren ihre Stärken aus. Folgende Tabelle gibt eine Übersicht über die Redundanzen, welche die Komprimierungsarten nach ihrer Arbeit hinterlassen und welche Eingangsredundanz sie behandeln. Dabei werden nur die Redundanzen auf Byteebene betrachtet.

Algorithmus	Eingangsredundanz	Ausgangsredundanz
Lauflängenkodierung	Symbolwiederholungen	Symbolverteilung Musterwiederholung
LZ77	Musterwiederholung	Symbolverteilung
LZSS	Musterwiederholung	Keine Redundanzen
LZ78	Musterwiederholung	Symbolverteilung
LZW	Musterwiederholung	Bei variabler Indexgröße keine weiteren Redundanzen
Huffmankodierer	Symbolverteilung	Keine Redundanzen
Arithmetischer Kodierer	Symbolverteilung	Keine Redundanzen
BWT	Musterwiederholung (reine Transformation)	Symbolwiederholung Ortsredundanzen
MTF	Ortsredundanzen (reine Transformation)	Symbolverteilung

Abbildung 2.62.: Redundanzein- und ausgaben der Komprimierungsalgorithmen

Aus der Tabelle 2.62 lässt sich ein Graph ableiten (Abbildung 2.63), der bei der Kombination der Algorithmen eine Hilfe sein kann. Je nach angenommener Eingangsredundanz kann man sinnvolle Algorithmenketten bilden. Dabei ist allerdings zu beachten, dass in der Praxis noch Größen wie Rechenaufwand und Zusatzinformationen der Algorithmen (zum Beispiel Kodierungsbaum bei Huffmankodierung) eine Rolle spielen. Als Grundkombinationen zum Testen empfehlen sich BWT>RLE, BWT>MTF>Huffman, RLE>LZW und LZ7X>Huffman.

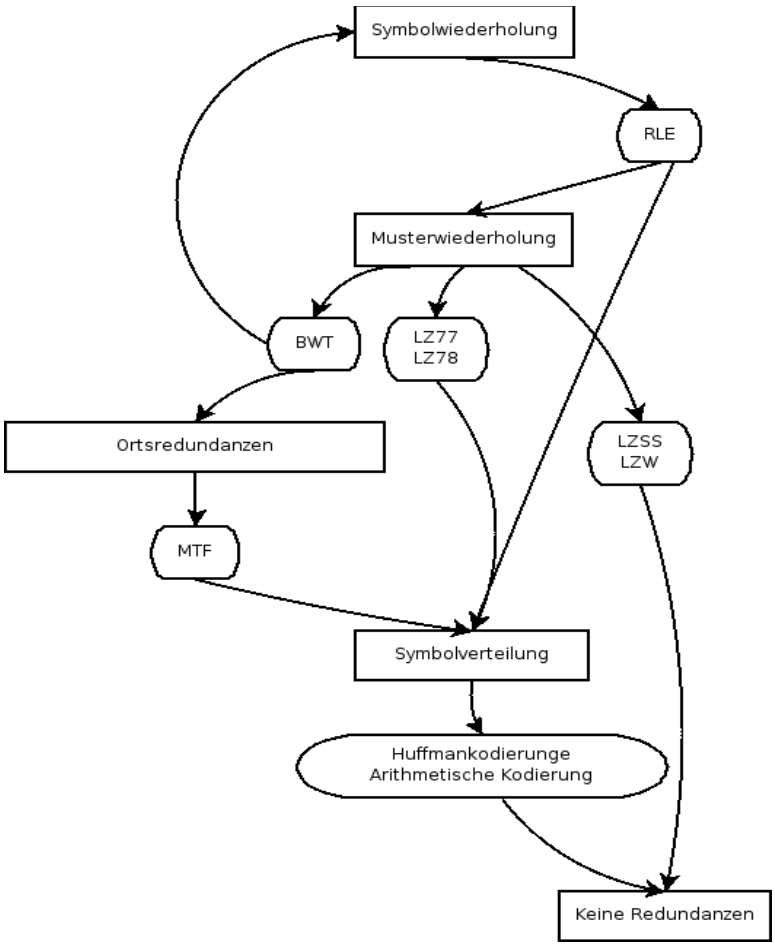

Abbildung 2.63.: Redundanzein- und ausgaben der Komprimierungsalgorithmen

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.

Algorithmus	Rechenaufwand	Zusatzinformationen bei kleinen Eingangsdaten
Lauflängenkodierung	gering	gering
LZ77	hoch	hoch
LZSS	hoch	hoch
LZ78	hoch	hoch
LZW	hoch	durchschnittlich
Huffmankodierer	sehr hoch	sehr hoch
Arithmetischer Kodierer	sehr hoch	sehr hoch
BWT	durchschnittlich	sehr gering
MTF	gering	keine

Abbildung 2.64.: Rechenaufwand und Zusatzinformationsanteil der Komprimierungsalgorithmen

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.

Der Rechenaufwand aller Verfahren, die ein Verzeichnis durchsuchen, kann mangels Arbeitsspeichers nicht sonderlich optimiert werden. Wird zusätzlich auf Bit-Ebene gearbeitet, steigt der Aufwand gegenüber Byte-Zugriffen deutlich. Bei der Huffman-Kodierung ist das größte Problem die Arbeit mit dem Kodierungsbaum, welche vollständig auf flachem Speicher abgebildet werden muss. Der Kodierungsbaum wird dabei für jeden Kodierungs-/Dekodierungsvorgang neu aufgebaut. Die Arithmetische Kodierung benötigt Fließkommaberechnungen, die auf eingebetteten Systemen in Hardware keine wirkliche Verbreitung haben. Sie in Software zu simulieren, erhöht den Rechenaufwand enorm.

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

Auf Reflex setzt das COPRA Framework (COmmunication PRocessing Architecture) auf. Dieses Framework basiert auf Protocol Procesing Stages (PPS), die sich zu Protocol Processing Engines (PPE) kombinieren lassen. Eine Stage repräsentiert in der Regel eine Protokollschicht, wie zum Beispiel Routing, Transport oder MAC. Durch dieses flexible Konzept ist es leicht möglich, Stages beliebig zu kombinieren. Eine PPE ist dann eine komplette Kommunikationseinheit, wie zum Beispiel Retransmission, Routing und Transport. COPRA stellt das untere Drittel des COCOS Projekts dar. Hier wird die gesamte Kommunikation realisiert, welche dann zum Beispiel von CHIPS (Convenient High-level Invocation Protocol Suite) für entfernte Methodenaufrufe benutzt wird. COCOS selbst benutzt wiederum CHIPS, um Operationen auf Gruppen zu ermöglichen. Die einzelnen Stages in COPRA kommunizieren hauptsächlich über den Stack und besitzen hierzu jeweils Methoden zum Empfangen (accept) und Senden (deliver) von Paketen. COPRA wird gefördert von der DFG (SPP 1140).

3.1.3 SERNET

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

emulieren [10]. Er stellt jeden Knoten in einem leichtgewichtigen Prozess auf Benutzerebene dar und wechselt präemptiv zwischen diesen. Das Radio wird dabei über eine spezielle SERNET-Schnittstelle angesprochen und emuliert einen „perfekten“ TDMA. Es werden eine grafische Darstellung und Ausgaben auf der Standardausgabe ermöglicht, was die Analyse und Fehlersuche bei der Entwicklung von Anwendungen erleichtert. Die Anwendungen laufen dabei in der Gastumgebung, welche in den unten durchgeführten Testläufen ein x86-System ist. Das Programm wird dabei für diesen Prozessor kompiliert und unterliegt dessen Einschränkungen und Möglichkeiten. Aus diesem Grund ist es nur sehr schwer, Voraussagen in Hinsicht auf den Rechenaufwand für die Realumgebung zu treffen.

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.
Bei der Komprimierung wird sich die Auswahl auf Lauflängen-, Huffman-, LZW-, MTF- und BWT-Kodierung beschränken. Diese decken ein breites Spektrum der theoretischen Verfahren ab und lassen so recht aussagekräftige Schlüsse in der Bewertung zu. Das Huffman-Verfahren bekommt bei den Entropiekodierungen den Vorzug vor der Arithmetischen Kodierung, da diese Gleitkommazahlen benutzt. Alternative Umsetzungen auf ganzzahlige Werte sind aus patentrechtlichen Gründen nicht umsetzbar und generell problematisch in 16 Bit-Umgebungen (siehe Analyse Kapitel 2.3.3.3). Bei den Wortkodierern erhält LZW den Vorzug, es verspricht die besten Ergebnisse der LZ-Familie, wenn es separat, also nicht in Kombination mit anderen Verfahren, genutzt wird.
Die Vielfältigkeit der Komprimierungsalgorithmen, ihre differenzierten Wirkungsszenarien und der Unterschied im Aufwand der zu erwartenden Implementierungen legen ein flexibles Design nahe, welches Austausch und Kombination der Verfahren erlaubt. Die Komprimierungsalgorithmen werden sich daher von einer abstrakten Klasse ableiten, um nach außen eine gleichbleibende Schnittstelle zur Verfügung zu stellen. Diese Schnittstelle umfasst Methoden jeweils zur Kodierung und Dekodierung. Eingabeparameter sind dabei der Zeiger auf die Eingangszeichenkette und deren Länge, gefolgt von einem Zeiger auf einen Ausgabepuffer mit dessen Größe.
Durch die gemeinsame Schnittstelle wird allen Algorithmen eine Zeichenkette aus Byte übergeben, was ein gleichbleibendes Alphabet ergibt und die Verfahren auch in ihrer Wirkungsweise kompatibel hält. Anderenfalls kann es sein, dass ein Verfahren ein Word-basiertes Alphabet (16 Bit) nutzt, wodurch eine Redundanzart zwar für dieses Alphabet gut kodiert wurde, aus Sicht des Byte-Alphabets (8 Bit) allerdings kein gutes Ergebnis hervor bringt. Ähnlich ist es, wenn ein vorbereitendes Verfahren die Entropie einer Nachricht erhöht und dabei von einem Byte-Alphabet ausgeht. Betrachtet man diese Nachricht dann mit einem anderen Alphabet, kann es zu ungewollten Ergebnissen führen, bis hin zum Entropiewert 1. Folgendes Beispiel lässt dies gut erkennen. Werden Symbole als Bytewerte betrachtet, so kommt das Symbol "0xFF" gehäuft vor. Dagegen ist in der gleichen Nachricht mit Word-Symbolen betrachtet, keine Häufung eines Symbols zu finden (Abbildung 3.1).

0x0	0xFF	0x1	0xFF	0x2	0xFF	0x3	0xFF
0xFF		0x1FF		0x2FF		0x3FF	

Abbildung 3.1.: Nachrichtendaten mit verschiedenen Alphabeten betrachtet

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

Kodegröße und Speicherverbrauch werden entscheidende Kriterien sein, welche Funktionen in der Praxis den Weg auf die Arbeitsplattform finden. Diese unterscheiden sich zudem von Plattform zu Plattform. Wo einige Systeme mit großem Flashs-Speicher punkten, geizen sie auf anderer Seite mit der Größe des Arbeitsspeichers. Um für alle Szenarien anpassbar zu bleiben, wird die zu implementierende Schicht auch die Möglichkeit bieten, Teilfunktionen wegzulassen, zum Beispiel Aggregation ohne Timer oder nur Komprimierung ohne Aggregation. Da dies zur Kompilierungszeit bekannt ist, fiel die Entscheidung, dies über Präprozessor-Anweisungen konfigurierbar zu halten. So kann verhindert werden, dass unnötiger Kode übersetzt und eingebunden wird.

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

möglichst viele Zusatzdaten der vorangegangenen Schichten mit komprimiert werden können. Von unten sollte sie jedoch mindestens durch eine CRC-Schicht abgesichert werden, da schon ein „Bitdreher“ für die Komprimierung zu einer regelrechten Datenbombe werden kann.
Als Klassenname wurde „Compression“ für die kombinierte Schicht von Aggregation und Komprimierung gewählt (Abbildung 3.2).

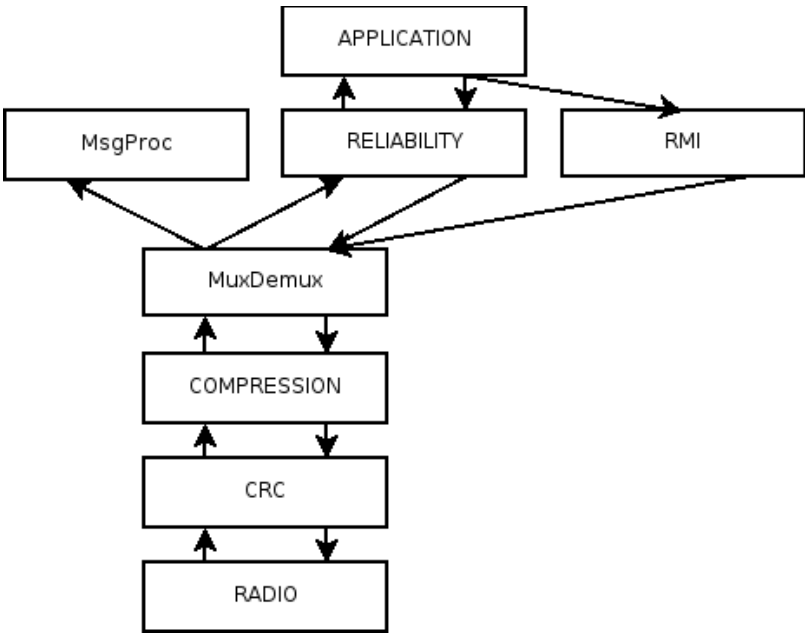

Abbildung 3.2.: Möglicher Aufbau einer PPE mit Compression-Schicht

3.6 Interner Aufbau der Compression-Schicht

Wird die Compression-Schicht mit ihrem vollem Funktionsumfang konfiguriert, entspricht sie dem folgenden Diagramm (Abbildung):

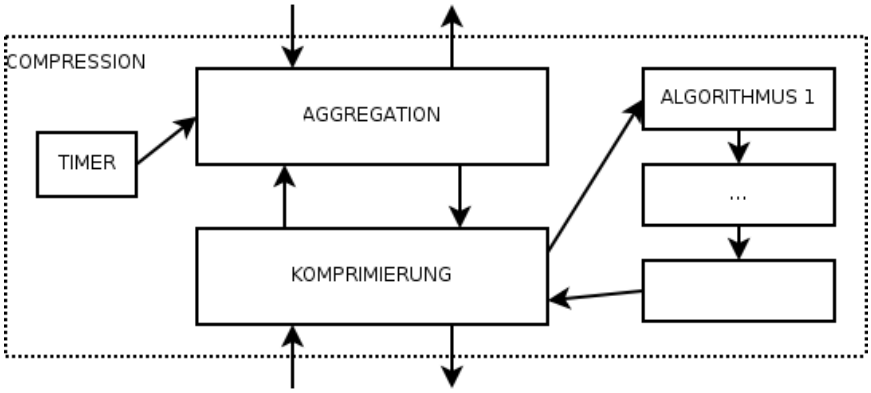

Abbildung 3.3.: Basisaufbau der Compression-Schicht

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

Zuerst werden die Komprimierungsalgorithmen implementiert. Diese können auch unabhängig von einem Netzwerk getestet werden. Da das Rahmenwerk zu Beginn dieser Arbeit noch nicht zur Verfügung stand und der Netzwerksimulator Schwächen in der Ausführung zeigt, war dies naheliegend. Dafür wurde ein Rahmenwerk aufgesetzt, das auf der Kommandozeile einen Packer realisiert, der intern die exakte Schnittstelle und Algorithmen nutzt, welche auch später in der Netzwerkschicht Verwendung finden sollen. So können die Algorithmen weit besser unter Stressbedingungen getestet werden, als das in einer Netzwerkumgebung oder auf dem Netzwerksimulator möglich wäre. Das Lokalisieren der Fehler und Analysieren der Abläufe gestaltet sich unproblematischer, da Daten jederzeit aus dem Bearbeitungsstrom umgeleitet werden können und das Programm auf das Wesentliche beschränkt bleibt.

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

(`Rle2Codec`) den Weg in die Praxis. Beide Klassen enthalten nur die vom Interface vorgegebenen Methoden und fallen in der Kodegröße gegenüber den anderen Algorithmen klein aus. Als zusätzliche Konstante enthalten sie lediglich Escape-Zeichen / Schwellwert (Abbildung 4.2).

```
class RleCodec : public CompressorCodec
{
[...]
      protected:
            enum escapechar_t {ESCAPECHAR = 123};
};
```

Abbildung 4.2.: Klassendeklaration von RleCodec

Aufwendiger ist die Implementierung des LZW-Algorithmus' (`LzwCodec`), da dieser ein Verzeichnis braucht, um bekannte Phrasen zu verwalten. Die Verzeichnisklasse (`Dictionary`) erfordert das Einfügen, komplette Löschen und das Suchen anhand des Indizes und der Phrase. Diese Funktionalitäten wurden allesamt auf einem flachen Feld realisiert, um den Speicheraufwand möglichst gering zu halten. Da dynamischer Speicher nicht zur Verfügung steht, muss die Größe des Feldes bereits zur Kompilierungszeit festgelegt werden. Ist das Verzeichnis voll, so wird es laut Algorithmus komplett geleert und mit der Kodierung normal fortgefahren. Die Symbole, die in die Ausgangsnachricht geschrieben werden, haben nicht mehr die Standardlänge von einem Byte (8 Bits), sondern von 9 Bits. Dies erfordert Schreib- und Lesezugriffe mit variabler Bitlänge, was über eine separate Klasse namens `BitField` gelöst wird. Diese bekommt den Ausgabespeicher als Initialisierungsparameter übergeben und kann auf diesem sequentielle Lese- und Schreibzugriffe mit variabler Bitlänge vornehmen. Neben diesen Zusatzklassen erfolgen die Kodierungsabläufe komplett in den Methoden der Interfaceklasse `CompressCodec`. Die maximale Größe des Verzeichnisses wird über eine Konstante der Klasse gesetzt (Abbildung 4.3).

```
class LzwCodec : public CompressorCodec
{
[...]
      protected:
            enum { DIC_SIZE = MAX_STACKMEMORY };
};
```

Abbildung 4.3.: Klassendeklaration von LzwCodec

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.

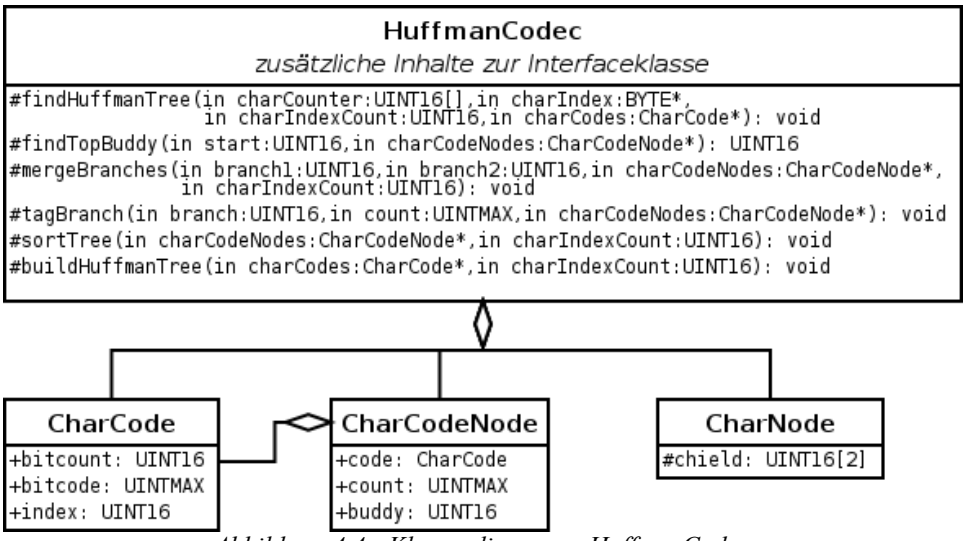

Abbildung 4.4.: Klassendiagramm HuffmanCodec

Das MTF-Verfahren, welches in Kombination mit BWT eine sinnvolle Ergänzung zur Huffman-Kodierung darstellt, erweist sich als weniger aufwendig. Einzig das aktuelle Alphabet muss zusätzlich im Speicher gehalten werden. Unter den implementierten Algorithmen ist MTF der Einfachste.

Komplexer stellt sich die Umsetzung von BWT dar. Neben dem zusätzlichen Speicherverbrauch durch eine Kopie der Nachricht ist ein Sortieralgorithmus notwendig, der diese in die definierte Reihenfolge bringt. Hierfür wurde ein wiederverwendbares

MergeSort erstellt, in der Vermutung, dass das Sortieren ein häufiger auftretendes Problem darstellt.
Durch das konsolenbasierte Rahmenwerk der Komprimierungsalgorithmen konnten umfangreiche Stresstests durchgeführt werden. Dafür wurden alle Dateien des lokalen Systems, die kleiner 150 Byte sind, durch zahlreiche Kombinationen von Komprimierungsverfahren geschickt. Dadurch konnte nicht nur die korrekte Funktion überprüft werden, sondern es ermöglichte auch eine Vorabschätzung über die Leistungsfähigkeit bei kleinen Nachrichten.

4.2 Einbindung der Komprimierungsalgorithmen in die Netzwerkschicht

Bei der Übertragung der getesteten Komprimierungsimplementierungen auf das Kommunikationsrahmenwerk ist die größte Herausforderung die Umsetzung der variablen Verkettbarkeit der Algorithmen. Dies soll schon zur Kompilierungszeit geschehen, aber einfach änderbar sein. Aus diesem Grunde wird auf Präprozessormakros zurückgegriffen, um die Definition in einer zentralen Konfigurationsdatei zu ermöglichen. Die genutzten Klassen, welche die Komprimierungsalgorithmen implementieren, werden als Objekte auf dem Stack erzeugt und Zeiger auf jene in ein Feld gespeichert. Die von der Netzwerkschicht aufgerufene Methode zum Komprimieren eines Pakets lässt die Paketdaten dann in einer Schleife durch die Kodierungsmethoden der Komprimierungsobjekte laufen.
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 der Arbeit des Rahmenwerks. Wird ein Paket oben in die Aggregationsschicht hinein gegeben, so heißt dies nicht mehr, dass

unten umgehend ein Paket wieder hinaus kommt. Kann ein zu sendendes Paket aus Platzgründen nicht zu den bisher gesammelten Paketen aggregiert werden, so kommt es zu einem "Flush". Die bisher verschmolzenen Pakete werden als ein Sendepaket an die nächste Schicht übergeben und das zurückgebliebene Paket bildet die Basis für eine neue Aggregation. Damit Pakete nicht ewig auf weitere Pakete zwecks Verschmelzung warten, gibt es die Möglichkeit einen „Flush“ explizit auszulösen. Dafür bietet die Aggregationsschicht die Methode `notify()` nach außen an, welche die unmittelbare Weitergabe der aggregierten Pakete auslöst. Dies kann optional auch ein integrierter Timeout übernehmen, der nach einer definierten Zeit einen "Flush" über diese Schnittstelle auslöst.

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).

```
#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).

```
[...]
#define COMPRESS_MERGE_ON
#ifdef COMPRESS_MERGE_ON
      #define COMPRESS_TIMEOUT_ON
#endif
#define COMPRESS_COMPRESS_ON
[...]
```

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.

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

Abbildung 4.7.: Detailkonfiguration in der CompressConfig.h

Die Kombination und Anzahl der Komprimierungsalgorithmen kann frei konfiguriert werden. Die Anzahl wird mit `COMPRESS_CODEC_COUNT` festgelegt. Anschließend legt man die Reihenfolge fest. Dies geschieht über ein Präprozessormakro. Es muss für die Anzahl der gewünschten Komprimierungsschritte jeweils ein Codec-Objekt erstellt werden, welche danach in das `codecs`-Feld eingesetzt werden (Abbildung 4.8).

```
[...]
#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;\
      codecs[1]   = &codec1;
#endif //COMPRESS_COMPRESS_ON
[...]
```

Abbildung 4.8.: Komprimierungskonfiguration in der CompressConfig.h

5 Bewertung

Um für die Bewertung Daten sammeln zu können, wurde auf die Simulationsumgebung SERNET zurückgegriffen. Die umgesetzte Komprimierungs- und Aggregationsschicht gibt dabei über die Standardausgabe an wichtigen Zeitpunkten Daten aus, die weiter analysiert werden. Die Energieeffizienz wurde auf Lego-RCX Robotern gemessen.
Der erste Fokus der Betrachtung fällt dabei auf die Menge der übertragenen Pakete und Daten. Daraus werden Erkenntnisse gewonnen, inwiefern die verschiedenen Konfigurationsbeispiele der Aggregationsschicht sich auf das Kommunikationsverhalten auswirken. Als Messwerte wurden Paketanzahl und übertragene Bytes pro kompletten Anwendungsdurchlauf gewählt. Um zufällige Abweichungen bei den Tests zu minimieren, werden mehrere Testdurchläufe durchgeführt und die erhaltenen Messwerte gemittelt.
Bei den Komprimierungsverfahren wird sich auf RLE>LZW beschränkt. Die Huffman-Implementierung scheidet aus, da die zusätzliche Übermittlung des Kodierungsbaumes, bei solch kleinen Nachrichten zu sehr ins Gewicht fällt. Als Alternativkombination wurde BWT>RLE eingesetzt.
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

2. Komprimierung BWT – RLE: Einfache Komprimierung der Daten mit der Kombination Burrows-Wheeler-Transformation und Lauflängenkodierung

3. Komprimierung RLE – LZW: Einfache Komprimierung der Daten mit der Kombination Lauflängenkodierung und Lempel-Ziv-Welch-Kodierung

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

Zeitzyklen sind abhängig von der Frequenz, mit der die Aggregationsschicht von außen getaktet wird. In der Konfiguration der Simulationen geschieht dies alle 250ms.
Die Beispielanwendung ist ein einfaches "Ping Pong" aller Netzknoten. Alle Knoten versuchen, nacheinander jeden anderen Knoten im Netz über ein "Ping Pong" zu erreichen.
Zu Beginn ist der Knoten mit der ID 0 Ziel aller anderen Knoten. Diese Situation ist vergleichbar mit dem praktischen Problem einer zentralen Datensenke, auf dem Weg zu der es zu starkem Netzverkehr kommt. Dies kann zu Flaschenhälsen in einem Netz führen, wo sich mehrere Pfade in einem Knoten treffen. Im Verlauf der Anwendung verteilen sich die angesprochenen Knoten, was zum Ende der Simulation zu sporadischem Netzverkehr führt. Dann sind einige Knoten bereits fertig mit der Anwendung, während wenige andere diesem Zustand noch entgegen streben. Damit wird ein breites Spektrum an möglichen Netzwerkszenarien abgedeckt.
Folgende Anwendungsszenarien kamen zum Einsatz (Abbildung 5.1 und 5.2):

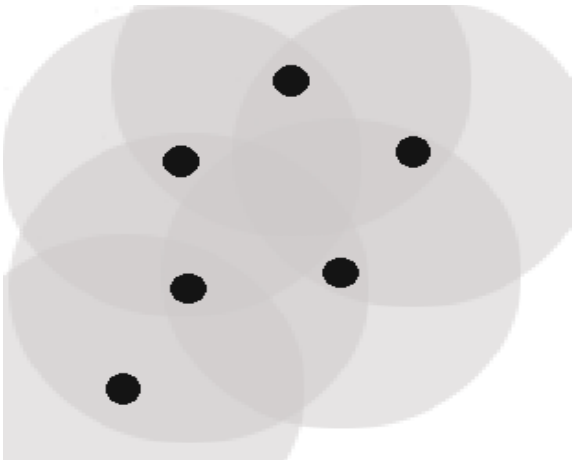

Abbildung 5.1.: Knotenanordnung in 6nodes

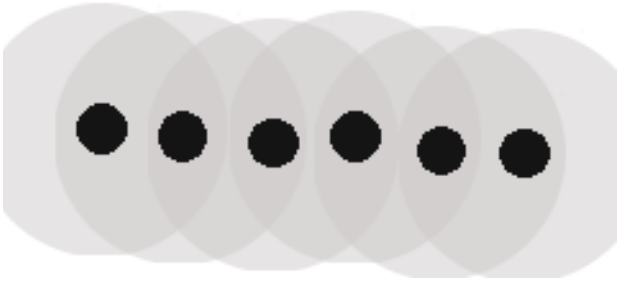

Abbildung 5.2.: Knotenanordnung in line6

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.
Im Vorfeld der Tests wurden anhand der Programmkonfiguration 1 Referenzmessungen durchgeführt und der Netzverkehr sporadisch mittels Dump-Schichten untersucht. Dabei wurden 2 Gruppen von Paketen identifiziert. Zum einen die Anwendungspakete, die inklusive Anwendungsdaten und Header vorangegangener Schichten zwischen 18 und 24 Byte groß waren, und zum anderen Pakete der Routingschicht, welche mit maximal 11 Byte vergleichsweise klein sind.

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

45 Byte als maximale Paketgröße ausgehender Pakete aus der Compression-Schicht wurde gewählt, da sie es in den meisten Fällen erlaubt, Pakete auch ohne Komprimierung wenigstens einmal zu aggregieren. Dennoch ist sie gering genug, dass Einflüsse einer erfolgreichen Komprimierung in den Messwerten deutlich werden.

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).

gesendete Pakete	566
gesendete Byte	11.606 Byte
durchschnittliche Paketgröße	20,5 Byte

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.

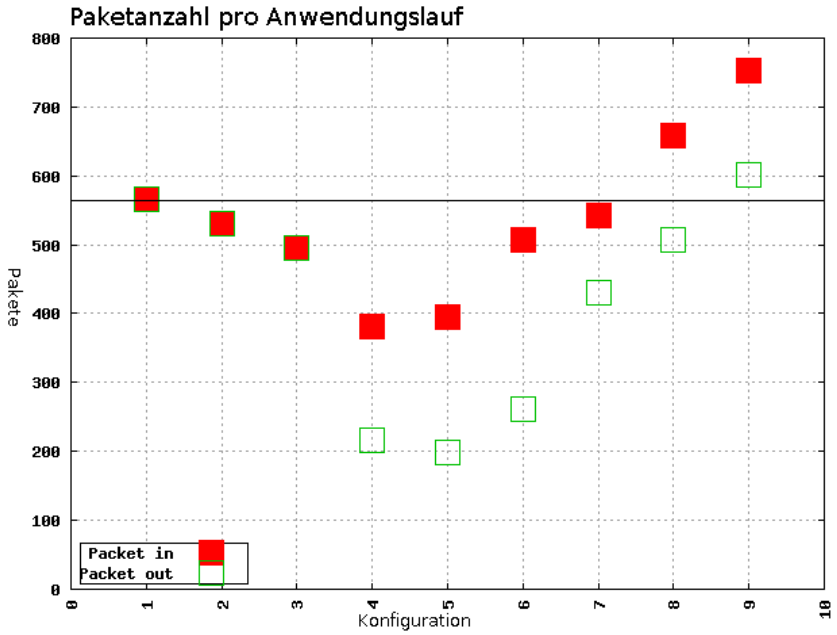

Abbildung 5.4.: Paketanzahl für Testfall line6

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 Anwendungspaketzahl 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):

gesendete Pakete	274
gesendete Byte	5.140 Byte
durchschnittliche Paketgröße	18,8 Byte

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%.

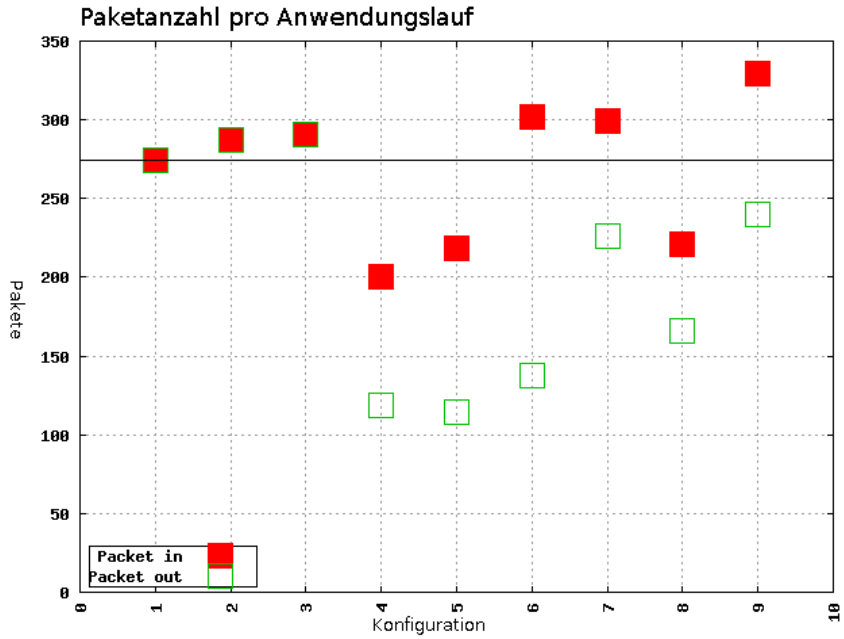

Abbildung 5.6.: Paketanzahl für Testfall 6nodes

5.1.2 Betrachtung des gesendeten Datenvolumens

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

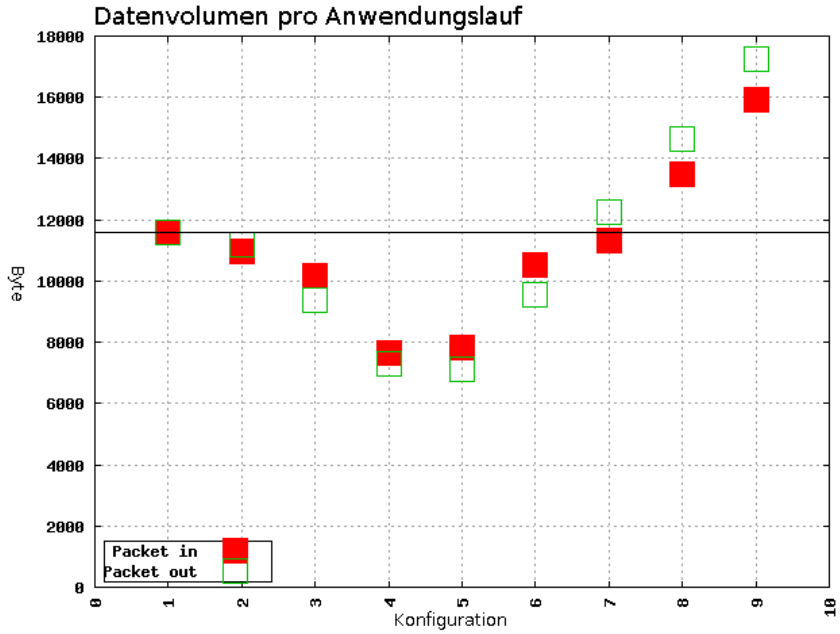

Abbildung 5.7.: Datenvolumen für Testfall line6

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.

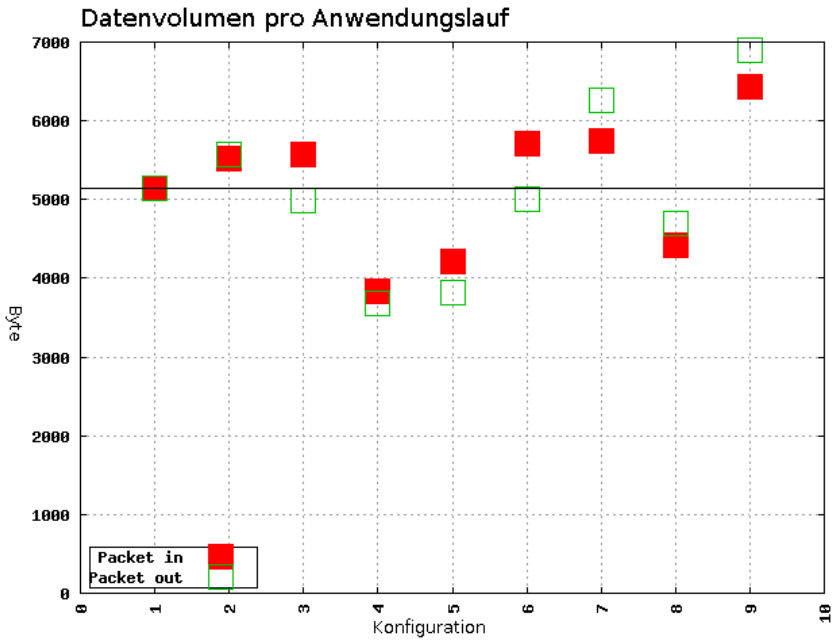

Abbildung 5.8.: Datenvolumen für Testfall 6nodes

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.

Für den praktischen Fall der "Easy-Radio"-Funkmodule [11] kann man anhand des Datenblattes überprüfen, wie viel man real an übertragenen Daten eingespart hat. Ein Charakter/Byte braucht bei 19.200 Baud 0,52 ms zur Übertragung. Dabei werden zusätzlich zu jedem Byte 2 Bits mitgesendet. Vor jedem Paket wird eine Präambel von 13,2 ms gesendet, was vergleichbaren übertragenen 31.73 Byte inklusive der Zusatzbits entspricht (etwas mehr als 25 übertragene Zeichen).

$$10\text{Bit}/19200\text{Baud} = 0{,}52\,ms$$
$$13{,}2\,ms/0{,}52\,Baud * (\frac{10}{8}) = 31{,}73\,Byte$$

Abbildung 5.9.: Formeln der real übertragenen Präambeldaten

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.

$$(zu\,sendende\,Byte\,inklusive\,Zusatsbits) + (Präambleaufwand) = real\,gesendete\,Byte$$
$$Messung\,1: (5.140\text{Byte} * 1{,}25) + (274 * 31{,}73\,Byte) = 15.119\text{Byte}$$
$$Messung\,8: (4.688\text{Byte} * 1{,}25) + (166 * 31{,}73\,Byte) = 11.127\text{Byte}$$
$$Messung\,5: (3.811\text{Byte} * 1{,}25) + (114 * 31{,}73\,Byte) = 8.381\text{Byte}$$

Abbildung 5.10.: Formeln des real übertragenen Datenvolumens

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.

$$(Byteersparnis) * 1000/Anwendungspakete = mögliche\,Byteoperationen/Paket$$
$$(15.119\text{Byte} - 11.127\text{Byte}) * 1000/221 = 18.063$$
$$(15.119\text{Byte} - 8.381\text{Byte}) * 1000/218 = 30.908$$

Abbildung 5.11.: Abschätzung der möglichen Byte-Operationen pro Aggregation

Der Rechenaufwand lässt sich aufgrund mangelnder Analysewerkzeuge nur theoretisch bestimmen. Beim reinen Aggregieren fallen lediglich 2 Kopiervorgänge der Paketdaten und der Verwaltungsaufwand des Betriebssystems beim Bearbeiten der Timeout-Aktionen an. Jeder Komprimierungsalgorithmus hat seine eigene Charakteristik in Bezug auf den zu erwartenden Rechenaufwand und die zu erzielenden Ergebnisse. Diese variieren zudem stark mit den zu verarbeitenden Eingangsdaten. Algorithmen, wie die Lauflängenkodierung und Move-To-Front, belasten den Prozessor und Speicher kaum, dagegen fallen LZW (Verzeichnisverwaltung), Huffman (Erstellung des Kodierungsbaum) und BWT (Sortieren der Nachricht) schwerer ins Gewicht. Genaue Aussagen können hierbei allerdings nur bei direkter Betrachtung im Zusammenhang mit einer Architektur und Anwendung getroffen werden.

5.1.4 Testszenario mit 25 Knoten

Da die bisherigen Tests nur mit 6 Knoten durchgeführt wurden, soll abschließend ein Testlauf mit einer Anordnung von 25 Knoten (Abbildung 5.12) betrachtet werden. Die Anwendung bleibt die gleiche, jedoch mit angepassten Werten.

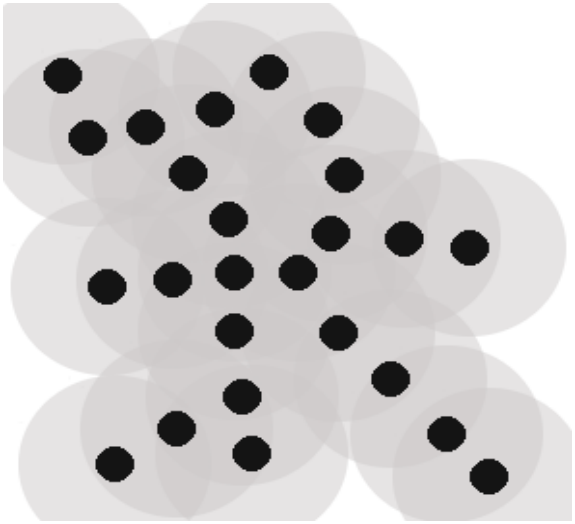

Abbildung 5.12.: Knotenanordnung in web25

- 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

Die maximale Paketgröße wurde angepasst, da durch die längeren Routen die Pakete voraussichtlich größer werden. Auf kurzen Routen wird dies dazu führen, dass mehr Pakete verschmolzen werden können, wenn im Wartezeitraum genug zur Verfügung stehen. Die Netztopologie gibt einen "Flaschenhals" im Zentrum vor. Nahe der Mitte sollte es im Praxisfall zu einer Knappheit der Sendekapazitäten auf MAC-Ebene kommen.

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.

Die Referenzmessung ergab 12.822 Pakete bis zum Ende der Simulation. Bei den Programmversionen 6 und 9 waren es jeweils 12.204 und 12.053 Pakete, die zum Versenden an die Aggregationsschicht übergeben wurden. Gesendet wurden bei den Versionen 6 und 9 jeweils nur 5.690 beziehungsweise 5.395 Pakete. In beiden Fällen konnte die Paketanzahl um mehr als die Hälfte verringert werden (Abbildung 5.13), dies bei nur geringer Erhöhung des übertragenen Gesamtdatenvolumens (siehe auch Messtabelle 5.17).

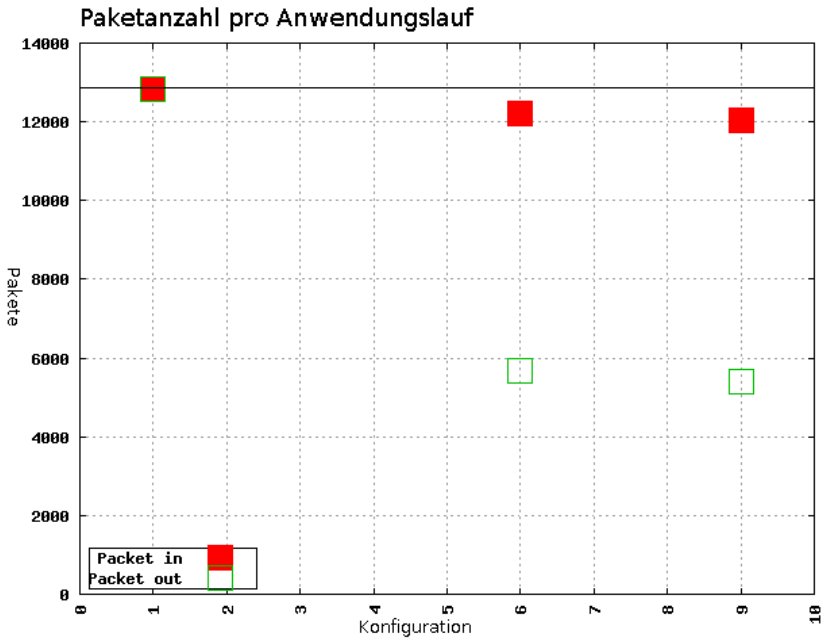

Abbildung 5.13.: Paketanzahl für Testfall web25

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).

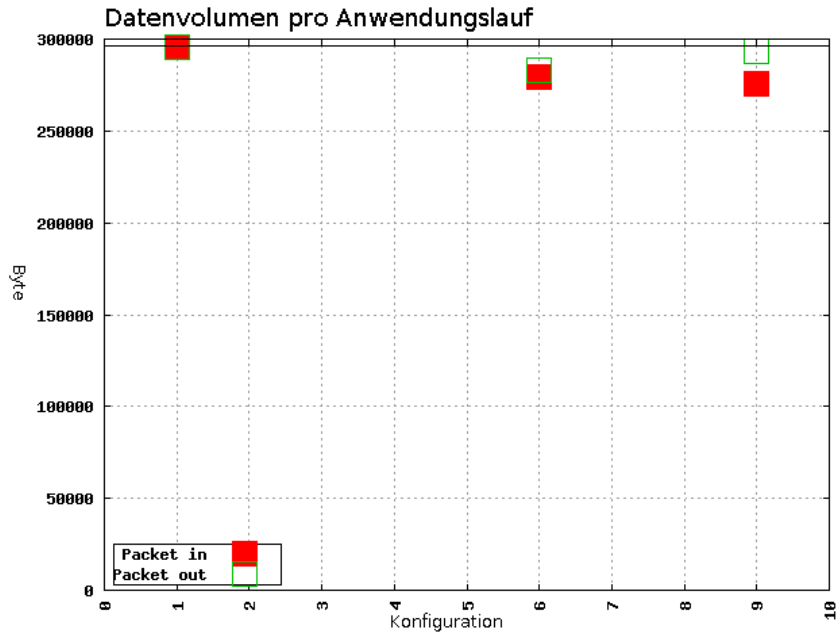

Abbildung 5.14.: Datenvolumen für Testfall web25

5.1.5 Messtabellen

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

Abbildung 5.15.: Komplette Messdaten für 6nodes

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

Abbildung 5.16.: Komplette Messdaten für line6

Programmkonfiguration	1	6	9
übergebene Pakete	12822	12204	12053
gesendete Pakete	12822	5690	5395
übergebene Byte	295303	279145	275810
gesendete Byte	295303	283084	293258

Abbildung 5.17.: Komplette Messdaten für web25

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.

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

Abbildung 5.18.: Messdaten der Kodegröße auf dem RCX

5.2.2 Betrachtung der Laufzeit und des Energieverbrauchs

Für die Betrachtung der Laufzeit wurde erneut die „PingPong“-Anwendung als Basis gewählt. Alle 500 Millisekunden wird dabei ein Ping-Paket von der Anwendung an die Netzwerkschichten übergeben. Dieses enthält die ID des Sendeknotens, eine Laufnummer und einen Zeitstempel. Die Aggregation arbeitet mit einem Timer, der nach 10 Sekunden einen „Flush“ auslöst, wenn dies nicht schon vorher durch andere Mechanismen geschehen ist. Die Pakete der Anwendung haben eine feste Größe von 6 Byte. Die maximale Paketgröße im Netzwerkprotokollstack beträgt 50 Byte, wobei die Aggregation maximal 45 Byte sammelt, um den nachfolgenden Netzwerkschichten ausreichend Platz zum Arbeiten übrig zu lassen.

Als Maß der Laufzeit werden Ticks des Prozessors gezählt, welche direkt aus dem Zählregister 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.

	1.) Aggr.	2.) Aggr. / RLE>LZW	3.) Aggr. / BWT>RLE
Durchschnittliche Laufzeit `Compression`	2.601	2.214	2.532
Durchschnittliche Laufzeit Sendeseite pro gesendetem Sammel-Paket	15606	28.782	32.916
Durchschnittliche Laufzeit Empfangsseite pro empfangenem Sammel-Paket	329	1.790	1.760
Durchschnittlich aggregierte Pakete	6	13	13

Abbildung 5.19.: Messdaten der Laufzeit auf dem RCX in Ticks

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 Größenangabe 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, die Ticks 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):

$$Prozessorzyklen\,komplette\,Schleife / Ticks\,komplette\,Schleife = Prozessorzyklen\ je\,Tick$$
$$1000 * 30 / 423 = 70{,}92$$

Abbildung 5.20.: Relation eines Ticks zu Prozessorzyklen

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):

$$Präambeleinsparung * (aggregierte\,Pakete - 1) = Einsparung\ je\,Sendevorgang$$
1. $31{,}73\,Byte * 5 = 158{,}65\,Byte$
2. $31{,}73\,Byte * 12 = 380{,}76\,Byte$
3. $31{,}73\,Byte * 12 = 380{,}76\,Byte$

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 benötigten Ticks berechnen (Abbildung 5.23).

$n : Anzahl\,der\,Ticks$
$t : realle\,Laufzeit\,s$
$e : Arbeitsleistung\,mWs$
$n * 70{,}92 / 16.000.000 = t$
$t * 100\,mW = e$

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

Das Funkmodul „EasyRadio“ verbraucht im Sendemodus 125 mW und auf Empfangsseite 105 mW [11]. Ein übertragenes Byte braucht bei 19200 Baud 0,52 ms. Die Arbeitsleistung in Abhängigkeit von den gesendeten Byte berechnet sich wie folgt (Abbildung 5.24):

n : Anzahl der Byte
t : Übertragungszeit s
e_t : Arbeitsleistung auf Sendeseite mWs
e_r : Arbeitsleistung auf Empfängerseite mWs
$n * 0.52\text{ms} * 1000 = t$
$t * 125\,mW = e_t$
$t * 105\,mW = e_r$

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).

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

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):

	1.) Aggr.	2.) Aggr. / RLE>LZW	3.) Aggr. / BWT>RLE
Durchschnittlich aggregierte Pakete	6	13	13
Durchschnittliche Energieeinsparung pro Paket	3,4 mWs	4,44 mWs	4,3 mWs
Durchschnittliche Energieeinsparung pro Paket in Prozent zum Normalfall	52%	68%	65%

Abbildung 5.26.: Durchschnittliche Energieeinsparung pro Einzelpaket

Für die gewählte Testanwendung mit der gegebenen Hardware ist eine deutliche Energieeinsparung beim Nachrichtentransport festzustellen. Allerdings zeigt dies nur die Möglichkeit einer positiven Energiebilanz in einem realen System auf. Größte Einsparungen brachten im Testfall die nicht gesendeten Präambeln, die im Verhältnis zur Nachrichtenlänge sehr groß ausfallen. Der Mehraufwand der Komprimierung konnte in beiden Fällen durch die Einsparung zusätzlicher Präambeln und durch mehr Pakete im Sammel-Paket ausgeglichen werden. Die bei der Komprimierung gesparten 33 Byte vielen dagegen eher weniger ins Gewicht.

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.

Der einfachste Fall wird in der folgenden Abbildung 5.27 dargestellt. Knoten A1 will ein Paket an A2, Knoten B1 an B2 senden. Beide Routen laufen über den Knoten C, der alle weiteren Knoten in seiner Reichweite hat. Während in den ersten beiden Schritten die Pakete noch jeweils einen eigenen Senderahmen belegen, um zum Knoten C zu gelangen, erfolgt im dritten Schritt das Senden beider Pakete innerhalb eines Sendevorgangs, da die Pakete auf C erfolgreich aggregiert werden konnten.

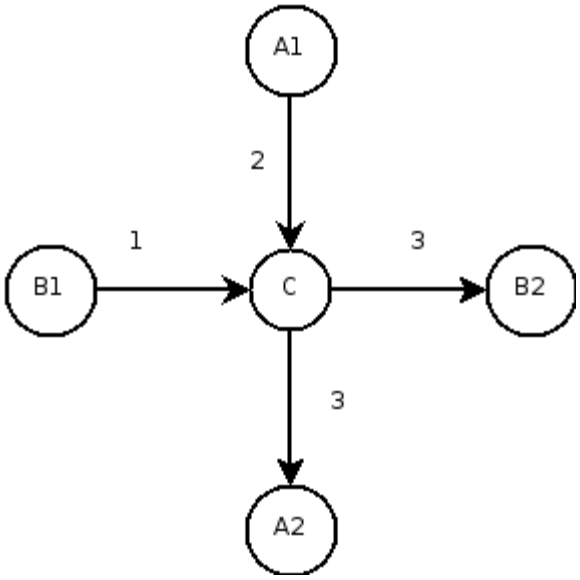

Abbildung 5.27.: Einfaches „Burst"-Verhalten der Aggregation

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).

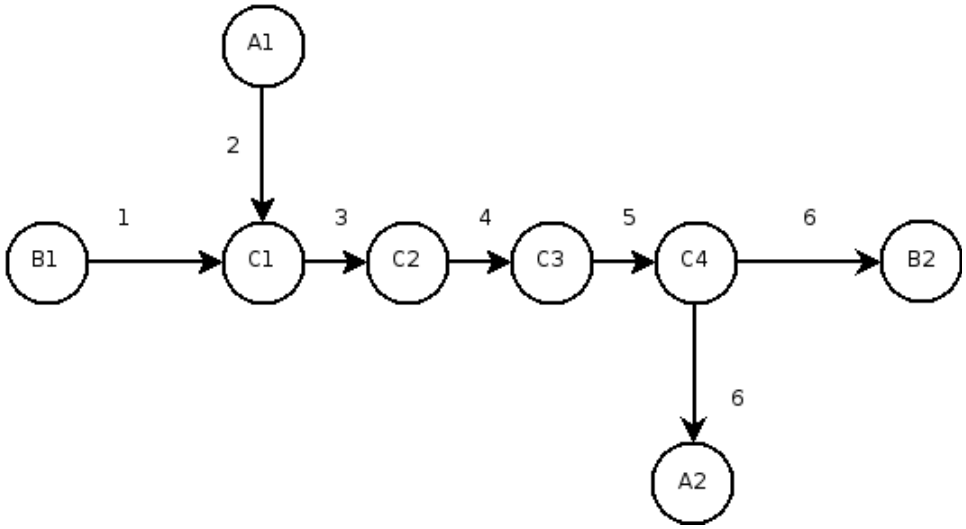

Abbildung 5.28.: „Burst"-Verhalten auf einem Teilpfad

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

gemeinsame Knoten, können Pakete verschmolzen werden. Dabei werden sie in einer Art "Burst"-Modus übertragen, solange sie sich einen Pfad teilen. Dies vermeidet Kollisionen und entlastet die MAC Schicht, da weniger Senderahmen benötigt werden.
Das Ziel der Energieeinsparung kann nur in bestimmten Fällen erreicht werden. Um Pakete zu verschmelzen, müssen im Sammelzeitraum genügend Pakete zum Senden übergeben werden. In einem Netz mit geringem Verkehr setzt dies eine Wartedauer voraus, welche den Wert der Latenz des gesamten Netzes deutlich erhöhen kann.
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 Aggregationstimeout 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.

Die vorgelegte Umsetzung sieht die Aggregation als völlig transparente Schicht (ausgenommen Zeitfaktor) vor, welche kein weiteres Wissen über ihre Nachbarschichten besitzt. Dies macht es jedoch nötig, dass Pakete, die sich einen gemeinsamen Pfad beim Routing teilen, auf jedem Knoten "entschnürt", vom Routing abgehandelt und neu aggregiert werden müssen. An dieser Stelle könnte ein solches Paket von verschmolzenen Teilpaketen separat geroutet werden. So müsste es erst am Ende des gemeinsamen Pfades entpackt werden. Dies setzt allerdings zahlreiche Kreuzabhängigkeiten der Netzschichten und zusätzliches Wissen der Netztopologie voraus.

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

[1] , Jochen Schiller & Achim Liers & Hartmut Ritter & Rolf Winter & Thiemo Voigt, ScatterWeb - Low Power Sensor Nodes and Energy Awaree Routing, , 2005

[2] , C.E.Shannon, "*A mathematical theory of communication*," Bell Syst. Techn. J., vol.27, July and Oct. 1948

[3] , Dean Ansley, ZSoft PCX File Format Technical Reference Manual, 1991

[4] , T.C.Bell, A unifying theory and improvements for existing approaches to text compression, Ph.D. Thesis, University of Canterbury, 1987

[5] , J.A.Storer & T.G.Szymansky, „*Data compression via textual substitution“*, Journal of the ACM, 1982

[6] , J. Ziv and A. Lempel, "*Compression of Individual Sequences via Variable-Rate Coding*," IEEE Transactions on Information Theory 24 (September 1978)

[7] , M. R. Nelson, „*LZW Data Compression“*, Dr. Dobb's Journal, October 1989

[8] , D.A.Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the I.R.E, 1952

[9] , M. Burrows and D. J. Wheeler, „*A block-sorting lossless data compression algorithms“,* Technical Report 124, Digital SRC Research Report, 1994

[10] , Andre Wallat, SERNet - Eine generische Emulationsplattform für eingebettete System mit drahtloser Kommunikation, , 2005

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

[12] , Hitachi, H8/300 Programming Manual,

[13] , Hitachi, H8/300 and H8/300L 8-Bit Microcontrollers, 1999