Fachhochschule Furtwangen
Studiengang Allgemeine Informatik

Diplomarbeit

Thema: Reduktion von Speicherbedarf und Transfervolumen in mobilen Anwendungen durch komprimiertes XML

Referent: Prof. Dr. Peter Fleischer

Korreferent: Dipl. Inform. Peter Rudolph

Vorgelegt im Sommersemester 2005
am 18. August 2005

von Arthur Miskolczi
Langstr. 36
79346 Endingen

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne unzulässige fremde Hilfe angefertigt habe. Die verwendeten Quellen und Hilfsmittel sind vollständig zitiert.

Furtwangen, den 18. August 2005.....

Arthur Miskolczi
Langstr. 36
79346 Endingen

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Mobile Lösungen

1.1.1 Stand der Technik

Mobile Geräte haben mittlerweile eine große Akzeptanz in der Bevölkerung und sind heutzutage kaum noch aus dem Alltag wegzudenken. Unter mobilen Geräten versteht man PDAs (Personal Digital Assistants), Notebooks, Smartphones, Wearable Computer und Mobiltelefone. Häufig werden sie zum Verwalten von Terminen oder Adressen eingesetzt, oft aber auch nur zum Telefonieren. Die Einsatzmöglichkeiten sind nahezu unbegrenzt, bedenkt man dass der Entwicklungsstand der mobilen Geräte den PCs nur wenige Jahre hinterher ist. Mal abgesehen von den Notebooks, sind Mobilgeräte trotz allem mit leistungsschwächeren Prozessoren und relativ wenig Speicher ausgestattet, zudem ist die Netzwerkbandbreite meist sehr gering. Der Datenaustausch findet üblicherweise auf zwei Arten statt, entweder über WAN[1] (GPRS[2], UMTS[3], ...) oder über LAN[4] (Wireless LAN, Bluetooth, ...). Die Kosten für die Benutzung von WAN Diensten wie z. B. GPRS oder ähnlichen Datenübertragungsstandards sind enorm und zwingen die Entwickler mobiler Softwarelösungen dazu, die zu übertragende Datenmenge klein zu halten.

1.1.2 ERP- und CRM Systeme im mobilen Einsatz

Damit Außendienstmitarbeiter auf Geschäftsprozesse zurückgreifen können, werden ERP[5]- und CRM[6]-Systeme im Zusammenhang mit mobilen Lösungen eingesetzt. Das Ziel ist, vorhandene Prozesse papierlos zu bearbeiten und für Beteiligte mit Hilfe eines bestehenden IT-Systems zeitnah zur Verfügung zu stellen. Im Allgemeinen geht es beim mobilen Einsatz von diesen Systemen darum, bisher manuelle Schritte durch vollständig integrierte mobile Lösungen zu ersetzen. Um einen besseren Einblick zu bekommen wird im Folgenden ein Beispiel aus

[1] Wide Area Network
[2] General Packet Radio Service. Ein Datenübertragungsstandard für GSM (Global System for Mobile Communication)
[3] Universal Mobile Telecomunication System
[4] Local Area Network
[5] Enterprise Ressource Plannig, z. B. Systeme wie SAP, Siebel, Navision
[6] Customer Relationship Management

der Praxis vorgestellt. Dies ist ein Projekt des Unternehmens PI-Data, in deren Hause auch diese Diplomarbeit geschrieben wurde. Die Beschreibung wurde von der Homepage[7] des Unternehmens übernommen und auf das wesentliche gekürzt:

> **Mobile Montage - SchwörerHaus**
>
> **Beschreibung**
>
> Unser Kunde SchwörerHaus ist einer der größten Fertighausherstellers Deutschlands. Ziel der hier vorgestellten Lösung war es, die Bearbeitung von Nachbestellungen während der Bauphase eines Hauses zu beschleunigen und den Bearbeitungsfortschritt für Monteure und Bauleiter transparenter zu machen. Dazu musste das in der Zentrale eingesetzte SAP R/3 um eine mobile Komponente erweitert werden.
> [...]
>
> **Lösung**
>
> Durch die strikte Trennung von Datenhaltung und Benutzerführung und dem intensiven Einsatz von XML im zugrunde liegenden System, konnte der Entwicklungsaufwand trotz einer individuellen Benutzerführung gering gehalten werden.
> [...]

1.1.3 Was macht PI-Data

PI-Data entwickelt mobile Softwarelösungen. Zum Teil in enger Zusammenarbeit mit Partnern wie dem Fraunhofer Institut, T-Mobile und der Voigt Consulting AG. Dabei geht es um:

- die Integration bestehender manueller Geschäftsprozesse mittels mobiler Lösungen
- ohne sich auf bestimmte Mobilgeräte einzuschränken
- und der Anbindung an ein bestehendes ERP- oder CRM-System.

Bei der Entwicklung wird auf zwei wichtige Probleme geachtet: die nicht permanente Verbindung und die Benutzbarkeit des Mobilgeräts. Unter nicht permanent ist hier, die nicht 100%ige Mobilfunknetzabdeckung zu verstehen. Man muss sich nur Vorstellen, dass im obigen Beispiel ein Haus in einem kleinen Dorf im Schwarzwald gebaut wird, wo eine Funkverbindung sehr schlecht oder gar nicht vorhanden ist. Oder man stellt sich einen Außendienstmitarbeiter vor, der in seinem Wagen sitzt und zuvor in einer Tiefgarage geparkt hat. Nun möchte er vor dem Kundengespräch nochmals seine Unterlagen elektronisch durchgehen, hat aber keine Mobilfunkverbindung. Verständlicherweise ist es in beiden Fällen wichtig, auch offline arbeiten zu können. Ein weiterer wichtiger Aspekt dabei ist, die Benutzbarkeit des Mobilgeräts und der entsprechenden Software. Denn

[7] *Peter Rudolph:* `http://www.pi-data.de`

sonst wird der Außendienstmitarbeiter schnell dazu verleitet, auf dem ihm gewohnten Weg mit Kugelschreiber und Papier weiterzuarbeiten.

1.2 XML zur Datenübertragung

XML wird nicht nur von PI-Data, sondern von vielen anderen Unternehmen zum Datenaustausch verwendet. Warum aber gerade XML so massiv eingesetzt wird, soll durch das nächste Unterkapitel verdeutlicht werden.

1.2.1 Eigenschaften von XML

XML steht für "Extensible Markup Language" und sieht HTML[8] sehr ähnlich. Wie der Name schon sagt ist XML erweiterbar und besitzt deshalb eine flexible Datenstruktur. Dadurch ist die Beschreibung neuer Grammatiken mittels XML möglich. Der Plattform- und Betriebssystemunabhängigkeit wegen ist XML textorientiert. Zusätzlich entsteht dabei noch der Vorteil, dass es für Menschen leichter lesbar ist. XML Dokumente oder Daten sind daher nahezu selbsterklärend. Weitere wichtige Eigenschaften sind die Parsebarkeit und die Validierbarkeit[9], sowie die Möglichkeit der Beschreibung der Syntax und des Datenformats. Das Datenformat wird z. B. mit XML-Schema[10] beschrieben. Aufgrund dieser Eigenschaften ist der Informationsaustausch zwischen zwei verschiedenen Systemen, sowie die Weiterverarbeitung, relativ leicht. Einen kleinen Einblick in XML bietet der folgende Abschnitt.

Die Struktur von XML

XML muss "wohlgeformt" sein, das bedeutet, dass die Grundstruktur des XML-Dokuments der Spezifikation des W3C[11] entspricht, z. B. muss es einen Prolog und mindestens ein Element enthalten, zudem muss das ganze Dokument in einem Wurzelelement eingeschlossen sein. Zusätzlich zur Wohlgeformtheit gibt es noch die Gültigkeit. Ein gültiges XML-Dokument schließt die Wohlgeformtheit ein und hat eine existierende, verfügbare DTD[12] bzw. ein XML-Schema, das die konkrete Struktur bzw. die Datentypen beschreibt, und ist in Bezug auf dessen Regeln gültig.

XML besteht aus Elementen und Attributen, die beliebig tief verschachtelt werden können. Im Grunde ist es nichts anderes als eine Baumstruktur im Textformat. Folgendes Beispiel eines XML-Dokuments soll dies verdeutlichen:

[8] Hyper Text Markup Language

[9] Dadurch ist es möglich im Voraus zu sagen, ob ein XML-Dokument gültig ist

[10] Dient der Typdefinition eigener Datentypen mit Hilfe von XML. Ist zusätzlich sehr streng Definiert. Siehe: *C. M. Sperberg-McQueen, Henry Thompson:* `http://www.w3.org/XML/Schema/#dev`

[11] *World Wide Web Consortium:* `http://www.w3.org`

[12] Document Type Definition, dient der Strukturdefinition zum Teil auch der Definition eigener Datentypen. Aber nicht mit Hilfe von XML, sondern mit einer eigenen Definition. Siehe: *Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau:* `http://www.w3.org/TR/2004/REC-xml-20040204/#sec-prolog-dtd`

```
<?xml version="1.0">
<Adressbuch version="1.3">
    <Adresse ID="987654">
        <Name>
            <Vorname>Hans</Vorname>
            <Nachname>Mustermann</Nachname>
        </Name>
        <Wohnort>
            <Ort>Musterstadt</Ort>
            <PLZ>12345</PLZ>
        </Wohnort>
    </Adresse>
</Adressbuch>
```

1.2.2 Probleme mit XML

Aus dem obigen Beispiel ist zu erkennen, dass XML sehr gut zur Beschreibung strukturierter Datentypen geeignet ist. Allerdings entsteht bei großer Datenmenge sehr viel Redundanz. Für XML ist dies zwar aus Gründen der Wohlgeformtheit und der Gültigkeit wichtig, aber für die Datenübertragung von Nachteil, besonders bei schwachen, mobilen Geräten. Man muss sich nur eine Mobilfunkverbindung mit geringer Bandbreite vorstellen, über die eine große Menge von Daten versendet werden soll. Hierbei sind Übertragungsfehler völlig normal. Deshalb müssen viele Datenpakete mehrmals versendet werden und die Gesamtdauer der Übertragung verlängert sich deutlich. Für den Empfänger einer XML-Nachricht entstehen ganz andere Probleme: für ihn ist das Verarbeiten, also das Parsen des erhaltenen XML-Dokuments und das anschließende Binding der Daten, sehr aufwendig.

Ein weiterer Nachteil ist, dass Datensätze nicht direkt adressiert werden können. Benötigt man z. B. nur eine Adresse eines Adressbuch-Dokuments muss das ganze Dokument geparst werden. Der Aufwand und der Speicherverbrauch ist dabei sehr hoch.

1.3 Ziel der Diplomarbeit

Ziel der Diplomarbeit ist es, ein geeignetes Verfahren zur Kompression von XML auf mobilen Geräten zu finden, das das Datenaufkommen bei der Übertragung reduziert und zusätzlich für die Datenhaltung eingesetzt werden kann. Dabei ist zu beachten, dass die Struktur von XML möglichst aufrechterhalten wird, um XML in seiner gewohnten Form verarbeiten zu können, bzw. das Kompressionsverfahren so gestaltet ist, dass der zügigen Weiterverarbeitung nichts im Wege steht.

Kapitel 2

Kompressionsmöglichkeiten

2.1 Einführung in die Datenkompression

Bei der Datenkompression geht es um die Reduktion der benötigten Speicherkapazität von Daten. Diese können in beliebig kodierter Form vorliegen. Mit Hilfe eines Kompressionsverfahrens werden die Daten in eine andere, kompaktere, Darstellung umkodiert. Um die ursprüngliche Kodierung wieder zu erhalten müssen die Daten wieder dekomprimiert werden.

Kompressionsverfahren haben viel mit der Informationstheorie, dem Teil der Mathematik der sich mit Redundanzen beschäftigt, gemeinsam. In der Informationstheorie wird auch der Begriff der Entropie verwendet. In diesem Zusammenhang beschreibt die Entropie, wie viel Information in einer Nachricht steckt. Die Entropie gemessen in Bit gibt den Informationsgehalt einer Nachricht an.

Kodiert wird nach dem Aufbau eines Modells, das sich aus Daten und Regeln zusammensetzt. Das Modell entscheidet, welcher Code welchem Symbol zugeordnet wird. Grundsätzlich kann aber ein Code mit verschiedenen Modellen verwendet werden. In [37] beschreibt Mark Nelson, dass der Unterschied zwischen Entropie und Kodierung, also die Anzahl Bits, das Potential ist, welches der Komprimierung zur Verfügung steht. Dieses Potential möglichst komplett auszuschöpfen, ist das Ziel der Kompression. Dies ist aber nicht so einfach, weil im Allgemeinen die zu komprimierenden Daten ständig wechseln, z. B. könnte ein Benutzer Text, Binärdaten einer ausführbaren Datei oder einen digital abgespeicherten Spielfilm mit ein und demselben Verfahren komprimieren. Die dabei erzielten Kompressionsraten würden sich sehr unterscheiden. Eine effizientere Lösung wäre, eine geeignete Modellauswahl mit Kenntnis der Struktur der vorliegenden Daten. Diese Kenntnis verfeinert das Modell und macht somit das Kompressionsverfahren effektiver. Gleichzeitig wird die Kompressionsrate erhöht.

Diesen Vorteil des reduzierten Speicherbedarfs möchte man auch beim Versenden von Daten ausnutzen. Gerade bei dem in Kapitel 1 beschriebenen Szenario, mit einer geringen Bandbreite und enorm hohen Kosten für die Datenübertragungsmenge, ist die Kompression von Daten sehr attraktiv. Man sollte

aber nicht vergessen, dass die Kompression bzw. Dekompression mit weiteren, zeitintensiven Verarbeitungskosten verbunden ist. Besonders bei den schwachen Mobilgeräten ist dies stets zu beachten.

2.2 Kompressionsarten

Es gibt eine Vielzahl von Kompressionsarten. Die dabei verwendeten Verfahren sind sehr unterschiedlich, weil in den meisten Fällen das Wissen über die Struktur der zu komprimierenden Daten ausgenutzt wird, um eine höhere Kompressionsrate zu erlangen. Dennoch lassen sich die Kompressionsverfahren in zwei Klassen einteilen, die verlustbehafteten und die verlustfreien Kompressionsverfahren.

2.2.1 Verlustbehaftete Kompressionsverfahren

Die verlustbehafteten Kompressionsverfahren werden meist bei digitalisierten, ursprünglich analogen Daten angewandt, z. B. bei Bildern, Filmen oder Sprach- bzw. Musikaufnahmen. Sie komprimieren Daten mit einem gewissen Teil an Informationsverlusten. Je höher die Verluste desto höher ist auch die Kompressionsrate. Diese Technik wird bewusst eingesetzt, weil dabei sehr hohe Kompressionsraten entstehen und die Verluste vom Benutzer kaum bemerkt, und daher toleriert, werden. Ein Beispiel wäre das Abschneiden der Frequenzen höher 20 kHz bei Audiodaten, da sie vom menschlichen Ohr nicht wahrgenommen werden können. Die erlaubt die Abspeicherung eines geringeren Frequenzspektrums, wofür weniger Bytes benötigt werden.Bekannte verlustbehaftete Kompressionsverfahren sind z. B. JPEG[1], MPEG[2], mp3[3] usw.

Man sollte stets beachten dass beim Einsatz eines verlustbehafteten Verfahrens, aus den neu entstandenen Daten nie wieder das Original hergestellt werden kann. Beim Dekomprimieren entsteht immer nur eine gewisse Ähnlichkeit zum Original. Aus diesem Grund scheiden die verlustbehafteten Kompressionsverfahren für eine Kompression von XML-Daten aus.

2.2.2 Verlustfreie Kompressionsverfahren

Verlustfreie Kompressionsarten arbeiten grundsätzlich auf zwei verschiedene Weisen. Auf Grundlage der Wahrscheinlichkeiten der auftretenden Symbole der zu komprimierenden Daten oder mit Hilfe von Tabellen. Beim Komprimieren mit Hilfe eines verlustfreien Verfahrens, sind die neu entstandenen Daten immer äquivalent zu den Quelldaten. Es kann immer wieder das Original hergestellt werden.

[1] JPEG: Joint Photographic Experts Group. Ein Kompressionsverfahren zum Kodieren von digitalen Bilddaten

[2] MPEG: Moving Picture Experts Group. Ein Kompressionsverfahren zum Kodieren von digitalen Filmen

[3] mp3: MPEG Layer III. Ein Kompressionsverfahren zum Kodieren von digitalen Audiodaten

Statistische Verfahren

Hierfür werden die Wahrscheinlichkeiten der auftretenden Symbole benötigt. Dazu kann eine statische Wahrscheinlichkeitstabelle benutzt werden. Diese gibt die Wahrscheinlichkeit für das Auftreten eines Symbols im ganzen Dokument an. Die zu Beginn aufgebaute Wahrscheinlichkeitstabelle, wird zusätzlich zu den komprimierten Daten abgespeichert, damit das Dekomprimierungs-Programm die Daten wieder herstellen kann.

Möchte man die Wahrscheinlichkeiten der Symbole besser vorhersagen, weil man z. B. weiß dass nach einem ”q” mit hoher Wahrscheinlichkeit ein ”u” folgt und damit weniger Bit zum Kodieren benötigt werden, muss man ein Modell höherer Ordnung verwenden. Die Ordnung gibt an, wie viel der zuletzt eingelesenen Symbole des Datenstroms für die Berechnung der Wahrscheinlichkeit des folgenden Symbols berücksichtigt werden. Ein Modell der Ordnung 0 würde keine vorhergehenden Symbole berücksichtigen und wäre somit ein statisches Modell. Ein Modell der Ordnung 1 würde im vorherigen Bespiel z. B. mit einer 95%igen Wahrscheinlichkeit ein ”u” nach einem ”q” erwarten und könnte es somit z. B. mit nur 2 Bit kodieren. Bei einem Modell der Ordnung 2 würden zwei vorhergehende Symbole zur Berechnung der Wahrscheinlichkeit des Folgesymbols berücksichtigt werden usw. Der Nachteil der Verwendung von Modellen höherer Ordnung ist der zusätzlich benötigte Speicherplatz für die Wahrscheinlichkeitstabellen. Man stelle sich vor, es sollte eine Datei komprimiert werden, die aus 8-Bit ASCII Zeichen besteht. Um die Wahrscheinlichkeitstabelle eines statischen Modells zu verwalten bräuchte man nun 256 Spalten. Bei einem Modell der Ordnung 1 bräuchte man aber schon 256 Tabellen mit je 256 Spalten und ein Modell 2ter Ordnung bräuchte 65536 Tabellen. Hieraus wird ersichtlich dass, diese Methode bei geringen Dateigrößen sehr ineffektiv ist da die Tabellen für das Dekomprimierungs-Programm mitgeführt werden müssen.

Die bessere Variante bietet das adaptive Verfahren. Es berechnet die Wahrscheinlichkeiten während die Daten vom Eingabestrom eingelesen werden und aktualisiert die Wahrscheinlichkeiten kontinuierlich. Damit dies funktioniert ist das Modell, das beim Komprimieren verwendet wird, mit dem Dekomprimierungs-Modell identisch. Der Nachteil entsteht am Anfang des Komprimiervorgangs. Zu diesem Zeitpunkt wurden noch keine bzw. nur wenig Symbole eingelesen und die Wahrscheinlichkeiten sind sehr schlecht. Symbole mit hoher Wahrscheinlichkeit gibt es kaum und dadurch müssen alle Symbole mit nahezu der gleichen, sehr hohen Anzahl Bits kodiert werden. Dies ändert sich aber rasch. Der Vorteil entsteht durch das identische Komprimierungs- und Dekomprimierung-Modell. Dadurch müssen die Wahrscheinlichkeitstabellen nicht mehr, den komprimierten Daten hinzugefügt werden, und es kann eine Menge Platz gespart werden, was sich positiv auf die Kompressionsrate auswirkt.

Unabhängig vom verwendeten Modell und dessen Ordnung wird meistens mit der Huffman- oder Shannon-Fano-Kodierung gearbeitet. Beide Verfahren sind sehr effektiv und kommen beim Kodieren dem Informationsgehalt einer Nachricht sehr nahe. Huffman hat bewiesen, dass seine Kodierung, unter den Kodierungen mit ganzzahligen Bits, nicht übertroffen werden kann. Somit ist die Huffman-Kodierung die bessere Wahl. Wie in [37] beschrieben wird, kann

eine sehr einfache Implementierung einer Huffman Kodierung, bei Anwendung auf Text, schon eine Kompressionsrate von etwa 30% erreichen.

Eine Verbesserung wäre die arithmetische Kodierung. Sie kommt bis auf ein Bruchteil eines Prozentes an den tatsächlichen Informationsgehalt einer Nachricht heran. Der Nachteil ist dass der Algorithmus langsamer ist. Die Kompressionsraten, die dabei erzielt werden, sind aber so hoch, dass dieses Verfahren trotzdem zum Einsatz kommt. Immerhin werden, wie in [37] beschrieben, bei Anwendung auf Text Kompressionsraten von etwa 40-70% erreicht. Lediglich auf langsamen Geräten, wo es nicht auf die höchste Kompressionsrate ankommt wird auf diese Kodierung verzichtet.

Tabellengesteuerte Verfahren

Diese Verfahren basieren auf der Ersetzung von Phrasen durch Tokens. Dabei wird viel Speicherplatz gespart und eine hohe Kompressionsrate erreicht. Die israelischen Forscher Jacob Ziv und Abraham Lempel beschäftigten sich in den 70ern mit Wort- und Zeichenhäufigkeiten um eine effektivere Art der Huffman Kodierung zu finden. 1977 veröffentlichten sie den Artikel "A Universal Algorithm for Sequential Data Compression" *in IEEE Transactions of Information Theory* der die LZ77 Kompression vorstellte und 1978 erschien der Nachfolger-Artikel "Compression of Individual Sequences via Variable-Rate Coding" der die LZ78 Kompression vorstellte. Diese beiden verschiedenen Kompressionsverfahren sind die Grundlage fast aller heutigen tabellengesteuerten Kompressionsverfahren.

Der LZ77 Algorithmus arbeitet mit einem gleitenden Fenster das den Ausschnitt des vorherigen Textes im Eingabestrom darstellt. Dieses Fenster bildet die Phrasentabelle und ist z. B. 4 kByte groß. Bei Redundanzen wird durch Position und Länge auf den vorherigen Text verwiesen. Andernfalls wird der Text unkodiert übernommen.

Der LZ78 Algorithmus arbeitet nicht mit einem gleitenden Fenster sondern mit einer theoretisch unbegrenzten Phrasentabelle. Bei Redundanz wird die entsprechende Phrase in die Tabelle aufgenommen und durch ein Token im Ausgabestrom repräsentiert. Diese Phrasentabelle ist nicht nur für ein gewisses Fenster wie beim Kompressionsverfahren LZ77 gültig, sondern für die gesamte zu komprimierende Datei. Somit werden auch weiter auseinander liegende Redundanzen erkannt und sparsamer kodiert.

Terry Welch's erweiterter LZ78 Algorithmus "LZW" ist sogar noch effektiver, allerdings ist er durch ein Patent geschützt, genauso wie viele LZ77 Varianten. Deshalb wird meist eine Variante des LZ77 Algorithmus in Kombination mit einer nachgeschalteten Huffman Kodierung wie bei den bekannten Zip Komprimierungs-Werkzeugen verwendet. Eine dieser Kombinationen nennt sich Deflate Algorithmus, und wurde so entwickelt dass alle patententierten Verfahren umgangen werden und dieses Verfahren somit für jedermann frei verfügbar ist. RFC-1950 [10] und RFC-1951 [11] beschreiben die Implementierung vom Deflate Algorithmus welche bei Gzip zum Einsatz kommt. Eine kurze Beschreibung der Arbeitsweise dieses Algorithmus ist in Kapitel 3.2.2 zu finden. Mit Gzip

kann Text laut Jeff Gilchrist's Kompressionstest[4] mit einer Kompressionsrate von 68% komprimiert werden.

2.2.3 Fazit

Ein verlustfreies Standardkompressionsverfahren wäre eine Lösung um XML zu komprimieren und anschließend zu versenden. Obwohl hohe Kompressionsraten damit erzielt werden können hat diese Lösung einen Nachteil. Das Problem ist, dass die Verarbeitung der komprimierten Daten zusätzliche Kosten mit sich bringt. Um ein XML-Dokument zu verarbeiten müssen die komprimierten Daten zuerst dekomprimiert werden, was einen zusätzlichen Aufwand zu dem ohnehin aufwendigen Parsen des XML-Dokuments erfordert.

2.3 Binäres XML

Eine andere Möglichkeit XML ohne zusätzlichen Aufwand zu Verarbeiten bietet eine binäre Kodierung des XML-Dokuments. Hierbei verspricht man sich eine schnellere Verarbeitung der XML-Daten gegenüber dem üblich eingesetzten Parser. Der Einsatz eines binären Formats ermöglicht auch eine kompaktere Darstellung der XML-Daten gegenüber dem Textformat, somit erreicht man auch einen gewissen Grad an Kompression. Für das Lesen und Schreiben eines binären XML-Formats werden spezielle Parser und Generatoren entwickelt die dieses Binärformat kennen und damit die kompakten XML-Daten verarbeiten können. Dabei verspricht man sich nicht nur eine kompaktere Repräsentation der XML-Dokumente sondern auch weniger CPU- und Speicher-Bedarf.

Das World Wide Web Consortion (W3C) hat die XML Binary Characterization Working Group gegründet die im Zeitraum März 2004 bis Ende März 2005 Untersuchungen bezüglich eines zukünftigen binären XML Standards machte. Dabei ging es darum Anforderungen, Eigenschaften, Use-Cases usw. für einen neuen binären XML Standard zu diskutieren. Des Weiteren wurden Erfahrungen der Mitglieder, entweder mit eigenen oder vorhandenen Lösungen in diesem Bereich, ausgetauscht. Die Untersuchung ergab dass eine binäre XML Lösung notwendig ist und in diesem Bereich etwas getan werden sollte. Weitere Informationen zu den Ergebnissen sind unter [58] zu finden.

In den nächsten Abschnitten werden die dort diskutierten Lösungen in einer kurzen Zusammenfassung vorgestellt. Diese Lösungen sind zum Teil sehr unterschiedlich und können folgendermaßen unterteilt werden:

1. Verfahren die auf Zeichenredundanzen basieren. Dazu werden hier Verfahren die auf einzelne Zeichen und Verfahren die auf Worte basieren, wie sie bei Markup-Sprachen vorkommen, siehe Beispiel weiter unten, zusammengefasst. Diese Verfahren nutzen überhaupt kein Wissen über XML aus. Der Vorteil einer Nutzung von Standardkompressionsverfahren ist die leichte Implementierung und die hohe Kompressionsrate welche damit erzielt werden kann. Nachteil ist dass hierfür zusätzliche Verarbeitungskosten entstehen die bei einem Mobilgerät nicht erwünscht sind.

[4] *Jeff Gilchrist:* `http://compression.ca/`

2. Verfahren die Wissen über XML ausnutzen, sei es Struktur, DTD, Schema oder ähnliches. Diese Verfahren nutzen sowohl das Wissen über XML als auch, zum Teil, Zeichenredundanzen aus. Diese Kategorie ist folgendermaßen unterteilt:

   (a) Infosetbasierte[5] Verfahren
   Durch die Verwendung einer geeigneten binären Repräsentation des XML-Dokuments, vor allem der Redundanzen wie z. B. Elementnamen, Attributnamen, Namespaces usw. ist es möglich diese wesentlich schneller zu verarbeiten als gewohntes XML in der Textdarstellung. Dies liegt an der weniger komplexen Struktur der Binärdaten. Hier müssen nicht so viele Zustandsinformationen wie beim normalen Textparser verwaltet werden. Der Vorteil ist, dass hiermit die gewohnte Textdarstellung von XML durch die Binärdarstellung, bei fast gleich bleibender Flexibilität, einfach ersetzbar ist. Wie in [43] beschrieben, ist der Nachteil gegenüber den schemabasierten Verfahren die allgemein ineffektivere Kodierung und die damit verbundene geringere Kompressionsrate.

   (b) Schemabasierte Verfahren
   Ein echtes Schemabasiertes Verfahren lässt Elementnamen, Attributnamen usw. weg, wenn sie aus dem XML-Schema wieder herstellbar sind und benutzt Datentypinformationen zum Kodieren. Somit kommt es dem Informationstheoretischen Minimum möglichst nahe. Diese Verfahren haben den Vorteil, dass sie nahe der Applikation bzw. des Bindingframeworks der entsprechenden Programmiersprache liegen und somit sehr schnell verarbeitet werden können. Ist die Datenreduzierung am wichtigsten, kann zusätzlich ein Standardkompressionsalgorithmus angewandt werden. Laut der Zusammenfassung [43] der Ergebnisse der Arbeitsgruppe des W3C kann durch diese Kombination die kleinste Nachricht für die meisten Applikationen erreicht werden, manche Mitglieder berichten sogar von einer Reduktion der Nachricht von mehr als 97%.

   (c) Hybridverfahren, eine Mischung aus Infoset- und/oder Schemabasierten Verfahren, mit redundanzbasierten Verfahren.

2.3.1 Zeichenredundante Verfahren

Prinzipiell zählen alle allgemein verwendbaren, verlustfreien Kompressionsverfahren zu den zeichenredundanten Verfahren. Deshalb werden alle Standardkompressionsverfahren inklusive einer Kodierung die speziell für Mobilgeräte entwickelt wurde zu dieser Kategorie hinzugezählt. Diese Kodierung heißt WBXML[6] und ist eine binäre Variante des WAP[7] Standards. Das folgende Beispiel soll verdeutlichen wie die Kodierung arbeitet:

[5] Infoset ist eine Abkürzung für XML Information Item Set. Infoset definiert die verschiedenen Arten von Informationen aus denen ein XML-Dokument besteht z. B.: XLink, XPath...

[6] WBXML: WAP Binary XML - Standard der entwickelt wurde um die benötigte Bandbreite beim Versenden von Nachrichten, für Mobilgeräte, zu reduzieren

[7] WAP: Wireless Application Protocol

```
<Auto>
     <Reifentyp>
     ...
     </Reifentyp>
</Auto>
...
<Auto>
     <Reifentyp>
     ...
     </Reifentyp>
</Auto>
```

Aus diesem Beispiel ist zu entnehmen dass, Verfahren die auf einzelne Zeichen basieren schlechter abschneiden als Verfahren die Worte, hier Tags, benutzen um Redundanzen zu beschreiben. Dies liegt an der Eigenart der Verfahren die auf einzelne Zeichen basieren. Das RLE[8] Verfahren kann nur in solchen Fällen effizient arbeiten, wenn gleiche Zeichen direkt hintereinander auftauchen. In gewohntem Text kommt bekanntlich eine direkt hintereinander folgende Wiederholung von Zeichen nicht so oft vor, dadurch ist klar dass hierdurch keine hohe Kompression erreicht werden kann. Oft wird, durch die benötigte Kodierung der Wiederholungen, sogar eine Vergrößerung der Originaldaten erreicht. Eine Verbesserung könnte man mit dem Einsatz eines statistischen Verfahrens mit einem Kodierungsmodell höherer Ordnung erreichen, das wiederum sehr speicherlastig arbeitet. Deswegen versucht man hier Verfahren die auf Tags basieren zu verwenden. Dies stellt eine Art der tabellengesteuerten Kompressionsverfahren dar, und zwar ohne gleitendes Fenster so dass, auch weit auseinander liegende Redundanzen die bei XML üblich sind, erkannt und mit möglichst wenig Bits kodiert werden. Ein gutes Beispiel liefert der Name des Wurzelelements, er taucht am Anfang und am Ende des Dokuments auf.

Im obigen Beispiel kommt `<Auto>` zwei mal vor und kann deshalb für das zweite Vorkommen einfach mit einer Referenz auf das erste Vorkommen von `<Auto>` kodiert werden. Hier können also 6 Bytes durch 1 Byte repräsentiert werden. Man kann sich vorstellen, wenn dies für alle auftretenden Tags einer Datei gemacht wird, so kann hiermit schon sehr leicht eine relativ gute Kompressionsrate erreicht werden.

In dieser Diplomarbeit werden alle Standardkompressionsverfahren als gleichwertig behandelt. Sie sind relativ alt und deshalb gut ausgereift. Die meisten beruhen auf einer Kombination aus LZ77- und Huffman Algorithmus und erzielen bei gleichen Eingangsdaten sehr ähnliche Kompressionsraten. Eine Untersuchung bezüglich Kompressionsrate, Speicherverbrauch und Zeitbedarf ist in Kapitel 3.2 zu finden. Dort wurde ein Test aller gefunden Kompressionsverfahren, die unter Java zur Verfügung stehen, durchgeführt.

Standardkompressionsverfahren eignen sich gut zur Reduktion großer Datenmengen. Lediglich bzip2[9] hat eine besondere Erwähnung verdient, denn dieses Kompressionsverfahren eignet sich laut [4] besonders gut zur Kompression von

[8] RLE: Run Length Encoding
[9] *Julian Seward:* `http://www.bzip.org`

XML-Daten. Bzip2 arbeitet intern mit der Burrows-Wheeler-Transformation welche nichts anderes macht als die Zeichen so umzusortieren dass, die Wahrscheinlichkeiten für eine anschließende Huffman-Kodierung optimal sind. Durch diesen Präprozessor für den Huffman-Algorithmus erreicht bzip2 die höchste Kompressionsrate aller Verfahren dieser Kategorie und unterliegt ansonsten den gleichen Vor- und Nachteilen aller Standardkompressionsverfahren, für die gilt: je größer die Datenmenge, desto höher ist die Wahrscheinlichkeit dass Redundanzen häufiger auftreten und die Datenmenge prozentual gesehen besser komprimierbar ist. Trotzdem ist bzip2 durch die sehr aufwendige Burrows-Wheeler-Transformation nicht unbedingt für den Nachrichtenaustausch geeignet weil die Transformation sehr viel Daten als Input benötigt bis ein komprimierter Output generiert wird, d. h. bzip2 ist durch seine natürliche Latenz, die zusätzlich viel Speicher benötigt, nicht für interaktive Szenarien geeignet.

Aus dem Vortrag [34] von KDDI R&D Laboratories Japan beim W3C Binary XML Workshop ist zu entnehmen, dass bei Standardkompressionsverfahren wie z. B. Gzip, zusätzlich hohe Verarbeitungskosten entstehen und Standardkompressionsverfahren dadurch für Mobilgeräte generell unbrauchbar sind. Diese hohen Verarbeitungskosten treten sogar zweimal auf, nämlich zuerst beim Dekomprimieren der Nachricht und danach noch mal beim Parsen der Nachricht. Die Gründe der Ausschließung der Standardkompressionsverfahren für den mobilen Bereich sind:

- die schwache Hardwareausstattung der Mobilgeräte welche meistens mit einem Akku versorgt werden
- die hohe CPU-Belastung der Standardkompressionsverfahren
- die damit verbundene Gesamtlaufzeit des Mobilgeräts.

Außerdem setzt man beim mobilen Einsatz eher auf Verarbeitungsgeschwindigkeit als auf Kompressionsrate. Trotzdem muss das nicht den gänzlichen Ausschluss dieser Verfahren bedeuten, z. B. wird bei einem Kunden von PI-Data beim erstmaligen Einsatz des Mobilgeräts eine so genannte Initialladung von XML-Daten benötigt. Diese Initialladung besteht aus einer riesigen Datenmenge bei der so ein Standardkompressionsverfahren mit einer sehr hohen Kompressionsrate die richtige Lösung bieten kann. Hier kommt es nämlich weniger auf die Geschwindigkeit und mehr auf die zu versendende Datenmenge an, die zusätzlich noch Geld kostet wenn sie über ein Mobilfunknetz versendet werden muss.

Eine Studie über die Kompression von militärischen XML-Daten [5] besagt, dass Hybridverfahren generell besser sind als WinZip[10]. Wobei WinZip hier als Platzhalter für alle Standardkompressionsverfahren zu sehen ist. Zudem ist aus dieser Studie ebenfalls zu entnehmen dass WBXML die schlechtesten Kompressionsergebnisse der dort verglichenen Kompressionsverfahren erzielte. Leider wurde zu der Verarbeitungsgeschwindigkeit von WBXML keine Angabe gemacht.

[10] *WinZip International LLC:* `http://www.winzip.com/`

2.3.2 XML-Spezifische-Verfahren

Tabelle 2.1 wurde aus [43] übernommen und zeigt alle beim Binary XML- Workshop vorgestellten XML-Kompressionsverfahren, das jeweilige W3C Mitglied welches dieses Verfahren entwickelte und die entsprechende Arbeitsweise dieses Verfahrens. Ein Fragezeichen gibt an dass nicht genügend Informationen verfügbar waren um eine entsprechende Aussage machen zu können.

Lösung	Entwickler / Beschreibung	Infoset	Schema	Hybrid
Fast Web Services	specs)	✔	✔	-
BXML	Cubewerx	✔	-	✔
Xebu	University of Helsinki	✔	-	✔
XBIS	Dennis Sosnoski	✔	-	-
ESXML	High Performance Technologies	✔	-	-
Lionet	XBVM	✔	-	-
BinXML	Expway (MPEG-7 BiM mit Erweiterungen)	✔	✔	-
CBXML	IBM	✔	-	-
XimpleWare	XimpleWare	✔	-	-
Serialized DOM	Media Fusion	✔	-	-
Systematic XML Compression	Systematic Software Engineering	-	✔	-
CMF-B	L3 Communication	-	✔	-
TokenStream	BEA Systems	?	?	?
Tarari XML Tokenizer	Tarari	✔	-	-
XML-Xpress	Intelligent Compression Technologies	-	✔	-
XML Schema Tools	OSS Nokalva (ITU-T/ISO X.69? Spez.)	✔	✔	-
Xeus	KDDI	-	✔	-
Xfsp	Web3D	✔	-	✔

Tabelle 2.1: Übersicht der untersuchten XML-spezifischen Kompressionsverfahren beim Binary XML-Workshop des W3C

Alle Verfahren aus Tabelle 2.1 wurden, mit dem Hintergedanke des eventuellen Einsatzes dieses Verfahrens für die Lösung des Problems von PI-Data oder für eine mögliche Erweiterung des entsprechenden Verfahrens bei verfügbarem Quellcode, genauer untersucht. Nach der genauen Untersuchung dieser Verfahren kristallisierte sich eine grobe Vorauswahl heraus. Dabei waren folgende Kriterien von Bedeutung:

- Ist die Lösung in Java geschrieben und kann sie in die PI-Data Software integriert werden?
- Ist die Lösung für den Einsatz mit Mobilgeräten geeignet?
- Ist die Lösung speziell auf Datenkompression, Verarbeitungsgeschwindigkeit oder beides ausgerichtet?
- Ist es eine echte binäre XML Lösung oder ist es nur ein XML-spezifisches Kompressionsverfahren welches das zusätzliche Parsen nach dem Dekomprimieren der Daten erfordert?
- Welche Performanz bietet diese Lösung?
- Ist die Lösung verfügbar und ist diese individuelle Lösung auch woanders einsetzbar oder macht ihre Individualität das unmöglich?
- Würde es sich lohnen dieses Verfahren für den eigenen Einsatz, falls nötig, entsprechend zu erweitern?

- Ist die Lösung mit Lizenzgebühren verbunden oder durch Patente geschützt?

Alle Verfahren die diese Kriterien nicht erfüllen sind, mit dem wichtigsten Grund, der zur Nichterfüllung der Kriterien beisteuerte, in Tabelle 2.2 aufgelistet. Alle übrigen Verfahren werden als potenzielle Lösung für PI-Data angesehen und sind in Tabelle 2.3 aufgelistet. Sie werden in Kapitel 4.1 genauestens untersucht und in einer knappen Beschreibung vorgestellt. Wenn möglich werden Ergebnisse betreffend Kompression, Speicherbedarf und Verarbeitungszeit gezeigt. Aufgrund dieser weiteren Untersuchung sollen die besten Kandidaten der vorhandenen XML-spezifischen Kompressionsverfahren herausgefunden werden.

Lösung	Entwickler / Beschreibung	Grund
BXML	Cubewerx	keine Java Implementierung
ESXML	High Performance Technologies	erst zukünftig verfügbar
CBXML	IBM	nicht verfügbar
CMF-B	L3 Communication	nicht verfügbar (militärischer Standard)
Tokenstream	BEA Systems	keine Angaben des Entwicklers
Ximpleware	Ximpleware	kein Kompressionsverfahren, durch eine zusätzliche binäre Datei wird ein „Random Access“ bereitgestellt
Tarari XML Tokenizer	Tarari	nicht verfügbar
XML-Xpress	Intelligent Compression Technologies	nur MS-Windows Demo erhältlich
XFSP	Web3D	nur Beispielimplementation verfügbar
Serialized DOM	Media Fusion	nicht verfügbar
Systematic XML Compression	Systematic Software Engineering	nicht verfügbar
Lionet	XBVM	nicht verfügbar
Fast Web Services	Sun Microsystems (ITU-T/ISO X.69? specs)	erst zukünftig verfügbar
Xeus	KDDI	nicht verfügbar

Tabelle 2.2: Übersicht der XML-spezifischen Kompressionsverfahren die nicht mehr weiter betrachtet werden

Lösung	Entwickler / Beschreibung
BinXML	Expway (MPEG-7 BiM mit Erweiterungen)
Xebu	University of Helsinki
XBIS	Dennis Sosnoski
XML Schema Tools	OSS Nokalva (ITU-T/ISO X.69? specs)

Tabelle 2.3: Potentielle Lösungen eines XML-Kompressionsverfahrens für PI-Data

Kapitel 3

Standardkompressionsverfahren

Dieses Kapitel beinhaltet eine Messung aller frei verfügbaren Kompressionsverfahren die unter Java eingesetzt werden können. Um die Messung bewertbar zu machen wurde folgendermaßen vorgegangen:

3.1 Messung & Bewertung

Man sollte die notwendigen Messungen so durchführen, dass die Messergebnisse gegeneinander vergleichbar sind. Um dies zu Gewährleisten wird zunächst das verwendete System beschrieben, dies hat den Vorteil dass, externe Messergebnisse mit den hier vorgestellten besser vergleichbar sind. Was dabei beachtet, und wie dabei vorgegangen wurde wird weiter unten beschrieben.

Verwendetes System

Hardware:

Notebook - Clevo/Kapok 2850
Pentium III Mobile 900 MHz
256 MB RAM

Software:

SuSE Linux 9.1, Kernel 2.6.4-52-default
J2SE 1.4.2_03

3.1.1 Auswahl der Messdaten

Für XML liegt eine übliche Textkompression, mit Standardkompressionsverfahren, nahe. Um einen Benchmark aufzustellen der den Vergleich mehrerer Kompressionsverfahren ermöglicht wurde ein XML-Corpus[1] von James Cheney zusammengestellt. Dieser XML-Corpus beinhaltet verschiedenste XML-Strukturen

[1] *James Cheney:* `http://www.cs.cornell.edu/People/jcheney/xmlppm/xml-corpus.tar.gz`

sowie unterschiedlich große Dateien welche durch die Breite der Abdeckung als Repräsentation aller XML-Dokumente dient. Deshalb wird dieser XML-Corpus für die Tests der Standardkompressionsverfahren verwendet.

3.1.2 Messungen

Bei den Messungen ist folgendes zu beachten:

Kompressionsrate

Die Kompressionsrate wird üblicherweise auf zwei Arten angegeben. Zum einen als Verhältnis der vorherigen Datenmenge zu der komprimierten Datenmenge z. B. 3:1. Um eine bessere Übersicht zu gewährleisten ist dabei der Wert der komprimierten Datenmenge immer 1. Zum anderen wird die Kompressionsrate in Prozent angegeben. Die Berechnung dieses Wertes ist aus [37] übernommen und lautet folgendermaßen:

$$1 - \frac{komprimierteGroesse}{urspruenglicheGroesse} * 100$$

Interpretation der Prozentangabe:

- 0% keine Kompression
- 67% die Größe der komprimierten Datei beträgt 1/3 ihrer Originalgröße
- 100% perfekte Kompression (theoretisch)

Bei komprimierten Dateien die größer als die Originaldatei sind entsteht eine negative Kompressionsrate.

Zeitbedarf

Die Zeitmessung unter Java ist relativ einfach, mit der Methode

```
System.currentTimeMillis()
```

kann die aktuelle Zeit in Millisekunden abgefragt werden. Vor Begin und nach dem Ende der Kompression bzw. der Dekompression wird diese Methode abgefragt und bei der Ausgabe die Differenz gebildet.

Natürlich kann der Wert vom System abhängen. Je nach der sonstigen Auslastung, der Anzahl der laufenden Prozesse, Thread-Switches usw. kann der Messwert deutlich variieren. Deshalb muss darauf geachtet werden dass bei den Messungen die minimale Anzahl der parallel laufenden Prozesse eingehalten wird und das System nicht mit anderen Aufgaben ausgelastet wird. Dies sind zwar keine realen Bedingungen, wie sie beim späteren Einsatz vorkommen, aber um vernünftige, vergleichbare Messergebnisse zu erhalten wurde stets darauf geachtet.

Auf das Thread-Switch Verhalten von Java hat man leider keinen Einfluss, somit bleibt dies eine dem System überlassene Tätigkeit die trotzdem Zeitschwankungen in den Messungen verursachen kann. Hier hilft der Durchschnittswert einer großen Anzahl von Messungen, um ein besseres Ergebnis zu erhalten.

Speicherbedarf

Der Speicherbedarf ist bei Java Programmen nicht mehr ganz so einfach zu ermitteln wie der Zeitbedarf. Bei allen Java Programmen läuft ein GarbageCollector als eigenständiger Thread im Hintergrund. Dieser übernimmt die Freigaben des unbenutzten bzw. nicht mehr benötigten Speichers. Einen Eingriff auf das Verhalten des GarbageCollectors hat man als Programmierer nicht, selbst wenn man ein Objekt auf null setzt und der GarbageCollector weiß dass es nicht mehr benötigt wird, gibt es keine Garantie dass er sofort anläuft und den Speicher freigibt. Möchte man den GarbageCollector aber explizit aufrufen ist, das durch die Methode

```
System.gc()
```

möglich, allerdings nicht sehr hilfreich bei der Messung des Speicherbedarfs, da man nicht weis ob der GarbageCollector sofort losläuft oder irgendwann während der eigentlichen Verarbeitung für die man zusätzlich die Zeit misst.

Eine Möglichkeit zur Ermittlung des benötigten Speicherbedarfs besteht darin den Speicher vor der Kompression bzw. Dekompression abzufragen. Die Differenz zwischen den zurück gelieferten Werten der Methoden

```
Runtime.getRuntime().totalMemory()
```

und

```
Runtime.getRuntime().freeMemory()
```

beschreibt den momentan verwendeten Speicher des Systems. Bildet man die Differenz der zurück gelieferten Werte nach der Kompression bzw. Dekompression erneut, kann man zusätzlich die Differenz des Speicherbedarfs während der Kompression bzw. Dekompression ermitteln indem man die Differenz zwischen den Werten vor und nach der Kompression bzw. Dekompression bildet. Allerdings ist dieser Wert nicht sehr aussagekräftig, da man nicht weis in wie weit der GarbageCollector seine Arbeit verrichtet hat.

Eine andere Möglichkeit den Speicherbedarf zu ermitteln wäre der Einsatz des Java Profilers. Dieser benötigt aber selbst Rechenzeit und Speicher und verfälscht das Ergebnis ebenfalls. Setzt man den Profiler zur Messung des Speicherbedarfs getrennt von der Messung des Zeitbedarfs ein, kann der ermittelte Messwert des Speicherbedarfs zum Vergleich anderer Messwerte die auf die gleiche Art ermittelt wurden benutzt werden. Dies ist dann lediglich ein Wert der zu Vergleichszwecken in diesem Rahmen gilt.

3.1.3 Anzahl der Messungen

Um einen guten Durchschnittswert der Messungen zu erhalten, sollte man jede Messung mehrmals durchführen bzw. den Messvorgang automatisieren, so dass viele Messungen gemacht werden. Je größer die Anzahl der Messungen desto besser wird das Endergebnis. Eine geeignete Anzahl ist nicht ohne weiteres vorherzusagen, je nach Art der Messung und den möglichen Einflussfaktoren ist eine individuelle Mindestanzahl der Messungen zu ermitteln. Wird ein System bei einer Messung z. B. durch Faktoren die nicht beeinflussbar sind gestört, ist

der einzige Ausweg der, die Anzahl der Messungen so hoch einzustellen dass die Fehlerrate so gering wie möglich gehalten wird und einigermaßen brauchbare Durchschnittswerte erzielt werden können. Dieser statistische Wert kann dann zu Vergleichszwecken dienen.

3.2 Freie Kompressionsverfahren unter Java

Nach einer intensiven Recherche im Internet wurden mehrere Kompressionsverfahren die in Java implementiert sind gefunden. Viele von ihnen wurden nur zu Test- oder Demozwecken und daher nicht sauber implementiert. Einige von ihnen wurden so schlecht implementiert dass beim ersten Test klar wurde, dass eine weitere Untersuchung dieser Verfahren wenig Sinn macht. Dazu können unterschiedliche Gründe beigetragen haben, z. B.:

- die Kompressions- bzw. Dekompressionszeit: wenn ein Verfahren mehrere Minuten zur Kompression benötigte wurde es als unbrauchbar angesehen
- die Kompressionsrate: wenn ein Verfahren eine Kompressionsrate von weniger als 30% erzielte, wurde es als unbrauchbar angesehen

Da ohnehin nur wenige Kompressionsverfahren unter Java implementiert sind blieben nach dieser Auswahl nur noch wenige Verfahren übrig. Diese übrigen Verfahren werden zusammen mit den Messergebnissen in den folgenden Unterkapiteln vorgestellt. Sie sind nach ihrer grundlegenden Algorithmen unterteilt worden.

3.2.1 Statistische Verfahren

In dieser Kategorie wurden folgende zwei arithmetische Verfahren getestet:

1. Das arithmetische Verfahren[2] von Bob Carpenter. Es ist ab Java 1.2 lauffähig und unterstützt auch Modelle höherer Ordnung. Getestet wurden die Modelle der Ordnung 0 bis 8.
2. Das bijektive arithmetische Verfahren[3] von Tim Tyler. Es unterstützt nur ein Modell der Ordnung 0. Da die Implementierungen sehr unterschiedliche Geschwindigkeiten erzielen können wurde dieses zweite arithmetische Verfahren zusätzlich in die Tests mit aufgenommen. Leider gibt es keine Hinweise darauf ab welcher Java Version dieses Verfahren lauffähig ist.

Allgemeine Funktionsweise eines arithmetischen Verfahrens

Beim arithmetischen Verfahren betrachtet man das Intervall $[0; 1[$ der rationalen Zahlen. Voraussetzung ist eine Statistik der Symbolwahrscheinlichkeiten, die entweder mittels eines ersten Durchlaufs durch Auszählen ermittelt wurde, oder durch die Verwendung eines adaptiven Verfahrens das die Wahrscheinlichkeiten, während dem Kompressionsvorgang kontinuierlich anpasst, wie es bereits in Kapitel 2 vorgestellt wurde. Liegen die benötigten Symbolwahrscheinlichkeiten vor, wird das Intervall entsprechend der Verteilung der Symbole in Teilintervalle

[2] *Bob Carpenter:* `http://www.colloquial.com/ArithmeticCoding`
[3] *Tim Tyler:* `http://www.mandala.co.uk/biac/index.html`

eingeteilt. Für die Kodierung des ersten Symbols wird das Teilintervall in welches das erste Symbol fällt, als neues Gesamtintervall ausgewählt. Dieses neue Intervall wird wieder in Teilintervalle entsprechend der vorliegenden Symbolwahrscheinlichkeiten eingeteilt. Für das zweite Symbol wird das entsprechende Teilintervall in welches das zu kodierende Symbol fällt als neues Gesamtintervall ausgewählt, und entsprechend den Wahrscheinlichkeiten in neue Teilintervalle eingeteilt usw. Dies wird solange fortgeführt bis die zu komprimierende Nachricht zu Ende ist. Die komprimierte Nachricht ist nun allein durch die Angabe der Intervallgrenzen komplett kodiert. Daraus kann auch wieder die Originalnachricht hergestellt werden.

Damit man sich das besser Vorstellen kann wird nun ein Beispiel für die Kodierung des Textes "xml test" vorgestellt. Man betrachte zunächst Tabelle 3.1 welche die Verteilung der Zeichen und die Einteilung der Intervalle zeigt. Für das kurze Beispiel wurde die einfachere Variante gewählt und die Symbole bzw. Zeichen wurden ausgezählt. Die Wahrscheinlichkeit wurde ermittelt indem die Anzahl der Vorkommen eines Symbols durch die Gesamtzahl dividiert wurde. Bei der Einteilung des Gesamtintervalls in die einzelnen Teilintervalle ist es unwichtig welche Zeichen, welchem Intervall zugeordnet werden. Der Dekompressionsalgorithmus muss dabei nur genauso arbeiten.

Zeichen	Häufigkeit	Wahrscheinlichkeit	untere Grenze	obere Grenze
Leerzeichen	1	1/8	0,000	0,125
e	1	1/8	0,125	0,250
l	1	1/8	0,250	0,375
m	1	1/8	0,375	0,500
s	1	1/8	0,500	0,625
t	2	2/8	0,625	0,875
x	1	1/8	0,875	1,000

Tabelle 3.1: Wahrscheinlichkeitsverteilung & Intervallzuordnung der Zeichen

Nachdem die Symbolwahrscheinlichkeiten ermittelt wurden kann nun mit der Kodierung begonnen werden. Um hierfür ein besseres Verständnis zu erlangen betrachte man Tabelle 3.2 und Tabelle 3.3. Tabelle 3.2 zeigt den Kodierungsprozess und Tabelle 3.3 zeigt den Dekodierungsprozess des Beispieltextes.

Die neu gebildeten Intervallgrenzen werden wie folgt gebildet. Für die neue untere Grenze wird das Produkt der unteren Grenze, des zu kodierenden Symbols, und dem alten Intervallbereich mit der alten unteren Grenze summiert. Das Beispiel der Kodierung des ersten Symbols soll den Vorgang verdeutlichen. Ausgegangen wird von dem Gesamtintervall [0; 1[. Nun soll das erste Symbol "x" kodiert werden.

$$0,875 * 1 + 0 = 0,875$$

Für die neue obere Grenze wird das Produkt der oberen Grenze, des zu kodierenden Symbols, und dem alten Intervallbereich mit der alten unteren Grenze summiert.

$$1,0 * 1,0 + 0 = 0$$

Für das nächste Symbol wird dieser Vorgang wiederholt, und zwar so lange bis keine Eingangssymbole mehr vorhanden sind. Wie man später bei der Dekodierung sehen wird ist es ausreichend, sich die untere Grenze zu merken.

Zeichen	untere Grenze	obere Grenze	Bereich
	0,00000000	1,00000000	1,00000000
x	0,87500000	1,00000000	0,12500000
m	0,92187500	0,93750000	0,01562500
l	0,92578125	0,92773438	0,00195313
Leerzeichen	0,92578125	0,92602539	0,00024414
t	0,92593384	0,92599487	0,00006104
e	0,92594147	0,92594910	0,00000763
s	0,92594528	0,92594624	0,00000095
t	0,92594588	0,92594612	0,00000024

Tabelle 3.2: Kodierung der Zeichen "xml test" mit einem arithmetischen Kompressionsverfahren

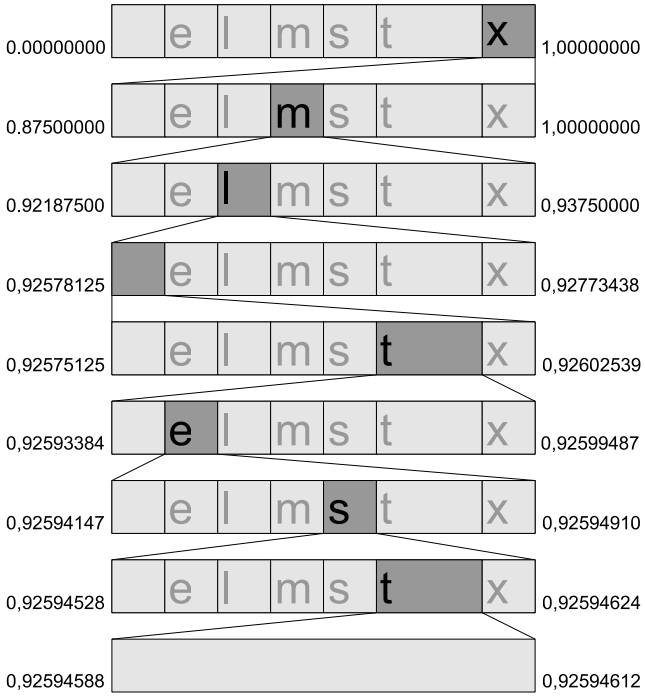

Abbildung 3.1: Kodierung der Zeichen "xml test" & schrittweise Intervallteilung

Abbildung 3.1 zeigt den Kodierungsprozess inklusive der Einteilung der Teilintervalle. Beachtet man die Intervallgrenzen, erkennt man sofort, dass das Intervall immer kleiner wird. Somit wären wir auch schon beim Problem des arithmetischen Verfahrens angelangt. Das Problem ist nämlich die Fliesskommazahlen-Berechnung, diese ist im Gegensatz zur Ganzzahl-Berechnung sehr aufwendig und lastet jeden Prozessor aus. Bei einer größeren Menge an Daten, müssen

eine Unmenge an Fliesskommazahlen-Berechnungen durchgeführt werden. Dadurch, dass die Intervallgrenzen immer näher aneinander rücken, wird der Nachkommastellenanteil der Intervallgrenzen so groß, dass eine Unterscheidung im Rechner, durch die begrenzte Stellenzahl, nicht mehr möglich ist. Um diesem Problem aus dem Wege zu gehen werden die Ziffern die sich nicht mehr ändern ausgegeben, in unserem Beispiel wäre das schon nach dem Kodieren des Zeichens "m" der Fall. Hier liegt die untere Grenze bei 0,921875 und die obere Grenze bei 0,9375, somit ist klar dass die darauf folgende Intervallgrenzen in diesem Bereich liegen müssen. Die 9 kann also ausgegeben werden. Nun wird die 0,9 von beiden Grenzen subtrahiert und die Intervallgrenzen werden mit 10 multipliziert. Das neue Intervall [0,21875; 0,375[passt zwar nicht mehr zu den vorherigen Intervallgrenzen, dies macht dem Verfahren jedoch nichts aus. Beim Dekodieren der komprimierten Nachricht betrachtet man dann immer soviel Stellen wie nötig, um eine Unterscheidung der Intervalle machen zu können. Eine bessere Möglichkeit, die vor allem auch Geschwindigkeitsvorteile bietet, ist es mit Ganzzahlen zu rechnen. Dazu verwendet man anstelle der Fliesskommazahlen Brüche mit einem gemeinsamen Hauptnenner. Die Zähler und Nenner welche durch Ganzzahlen repräsentiert werden, beschreiben nun die Fliesskommazahl. Beim Kodieren des nächsten Symbols werden die Brüche mit dem Hauptnenner erweitert, dabei werden die Zahlen immer größer genau wie beim Fliesskommaanzatz und deswegen werden auch hier Informationen ausgegeben.

Beim Dekodieren, siehe Tabelle 3.3, ist zu Sehen dass, die Angabe der unteren Grenze ausreicht um eine Nachricht dekodieren zu können. Dazu wird das Intervall des entsprechenden Kodes betrachtet und daraus mit Hilfe der Aufteilung der Wahrscheinlichkeiten aus Tabelle 3.1 dekodiert.

Code	Ausgabe	untere Grenze	obere Grenze	Bereich
0,92594588	x	0,875	1,000	0,125
0,40756702	m	0,375	0,500	0,125
0,26053619	l	0,250	0,375	0,125
0,08428955	Leerzeichen	0,000	0,125	0,125
0,67431641	t	0,625	0,875	0,250
0,19726563	e	0,125	0,250	0,125
0,57812500	s	0,500	0,625	0,125
0,62500000	t	0,625	0,875	0,250
0,00000000				

Tabelle 3.3: Dekodierung der Zeichen "xml test" mit einem arithmetischen Kompressionsverfahren

Für jeden Schritt der Dekodierung wird das Intervall angepasst. Nach der Dekodierung des ersten Symbols "x" wird die untere Grenze vom vorherigen Kode 0,92594588 abgezogen. Man erhält

$$0,92594588 - 0,875 = 0,05094588 \ .$$

Wird das Ergebnis durch die Breite des Intervalls dividiert erhält man den neuen Code.

$$\frac{0,05094588}{1,0-0,875} = 0,40756704$$

Bemerkung: In Tabelle 3.3 ist, aufgrund eines Rundungsfehlers des verwendeten

Tabellenbearbeitungsprogramms eine Zahl die sich in der letzten Nachkommastelle unterscheidet, zu sehen. Dies ist für die Erklärung der Arbeitsweise des Decoders jedoch nicht weiterhin tragisch.

Mit dem neu erhaltenen Kode 0,40756704 kann nun das nächste Symbol nach dem gleichen Prinzip dekodiert werden. Wie man sieht fällt dieser Kode in den Bereich der für das Symbol "m" festgelegt wurde. Wiederholt man diesen Schritt bis zum Ende der Nachricht, d. h. bis eine 0 als Kode auftaucht, ist man am Ende angelangt. Hier muss aber beachtet werden, dass eine 0 in den Bereich fällt der für das "Leerzeichen" festgelegt wurde. Um Schwierigkeiten zu vermeiden, sollte ein zusätzliches Zeichen, das das Ende markiert eingeführt werden.

Der Vorteil der arithmetischen Kodierung ist, dass sie wie die Huffman-Kodierung, in Verbindung mit Modellen höherer Ordnung eingesetzt werden kann. Diese Variante der arithmetischen Kodierung heißt PPM[4] und berücksichtigt schon eingelesene Symbole zur Voraussage des Folgesymbols. Wie in Kapitel 2 schon vorgestellt wurde, gibt die Ordnung die Anzahl der im Kontext berücksichtigten Symbole zur Vorhersage der Wahrscheinlichkeit des Folgesymbols an. Dadurch wird versucht die Wahrscheinlichkeitsverteilung auf den aktuellen Kontext anzupassen, damit Symbole effizienter kodiert werden können. Dies ermöglicht die Steigerung der Kompressionsrate mit steigender Ordnung. Der Nachteil dieser Variante ist jedoch die hohe Speicherauslastung des Verfahrens, denn für die Verwaltung der Kontexttabellen wird viel Speicher benötigt. Die lineare Erhöhung der Ordnung bringt einen exponentiell ansteigenden Speicherverbrauch mit sich.

Ergebnisse

Betrachtet man die Kompressionsrate der arithmetischen Verfahren (Abbildung 3.2 und Abbildung 3.3) verglichen mit dem Speicherverbrauch (Abbildung 3.6 und Abbildung 3.7) entsprechen die Messergebnisse den Erwartungen. Dies bedeutet je höher die Ordnung des Verfahrens ist, desto höher ist der Speicherverbrauch. Obwohl die Kompressionsrate ab dem Verfahren der Ordnung 4 kaum noch wächst, wächst der Speicherbedarf verglichen zur Ordnung 8 trotzdem noch bis zu einem Faktor von ca. 4 an. Trotzdem ist der Speicherverbrauch bei Ordnung 4 schon über einem MByte. Dies ist für Mobilgeräte schon viel zu viel. Betrachtet man die unterschiedlichen Implementierungen des arithmetischen Verfahrens in Abbildung 3.3, Abbildung 3.5 und Abbildung 3.7 ist zu erkennen dass in allen Messungen außer bei der Kompressionsrate das bijektive Verfahren besser ist. Bei der Zeitmessung ist es beim Komprimieren mit einem Faktor von fast 7, und beim Dekomprimieren mit einem Faktor von fast 5 besser als das andere arithmetische Verfahren. Beim Speicherverbrauch liegt das bijektive arithmetische Verfahren ebenfalls vorne, d. h. beim Komprimieren ist der Speicherverbrauch ziemlich gleich, beim Dekomprimieren jedoch benötigt das bijektive Verfahren 250 kByte weniger, was für ein Mobilgerät recht viel ist. Alles in allem scheint das bijektive Verfahren jedoch deutlich sauberer implementiert zu sein, da es bei der Ordnung 0, die beide Verfahren implementieren, im Zeitvergleich deutlich besser abschneidet.

[4] PPM: Prediction by Patial Matching

Das arithmetische Verfahren arbeitet wie man sieht sehr speicherlastig. Eine vernünftige Kompressionsrate kann erst ab der Ordnung 1 erreicht werden und selbst bei dieser Ordnung werden fürs Komprimieren schon ca. 250 kByte, und fürs Dekomprimieren schon ca. 600 kByte benötigt.

3.2.2 Tabellengesteuerte Verfahren

In dieser Kategorie wurden folgende Verfahren getestet:

1. LHA
2. JZlib
3. J2SE integrierte Kompressionsverfahren:
   (a) Gzip
   (b) Zip

LHA

Die hier verwendete Java Implementierung von LHA[5] wurde von Michel Ishizuka implementiert und ist ab Java 1.2 lauffähig[6]. LHA ist eine Variante des tabellengesteuerten LZSS Verfahrens und wurde ursprünglich von Haruyasu Yoshizaki entwickelt. Das LZSS Verfahren ist die erste einfache Implementierung des LZ77 Verfahrens. Das LZ77 verfahren wurde bereits in Kapitel 2 vorgestellt, und arbeitet mit einem gleitenden Fenster welches die Phrasentabelle des Eingabestroms bildet. Redundanzen können einfach durch Position und einer Längenangabe kodiert werden. Zur Verdeutlichung betrachte man Abbildung 3.8.

LZSS wurde 1982 von James Storer und Thomas Szymanski entwickelt. Dabei wurde versucht das ursprüngliche LZ77 Verfahren zu verbessern weil es etliche Effektivitätsprobleme hatte, und alles andere als einfach zu implementieren war. Zum einen wurde im laufenden Textfenster, der Phrasentabelle, eine Baumstruktur untergebracht um Übereinstimmungen schneller Suchen zu können. Zum anderen wurde die Tokenausgabe verbessert. Beim LZ77 Verfahren musste man für jedes neue Zeichen ein Offsetdummy, eine Länge von 0 und das Zeichen ausgegeben. Diesen Overhead versuchte man beim LZSS Verfahren zu reduzieren indem man an der Stelle eines zu kodierenden Zeichens 1 Bit voranstellte, welches entweder ein folgendes Offset-Längenpaar oder ein einzelnes Byte angibt. Dies erhöht die Kompressionsrate enorm. LHA beinhaltet mehrere Varianten des LZSS Verfahrens mit nachgeschalteter Huffman-Kodierung. Eine Übersicht der unterstützten und getesteten LHA-Varianten bietet Tabelle 3.4 welche von [28] übernommen wurde.

Ein vereinfachtes Beispiel des LZSS Verfahrens, für den Text "XML und XML-Schema" könnte folgendermaßen aussehen. Der Kodierungsvorgang geht dabei den Eingangsstrom Zeichen für Zeichen durch. Jedes eingelesene Zeichen

[5] *Michel Ishizuka:* `http://homepage1.nifty.com/dangan/en/Content/Program/Java/jLHA/LhaLibrary.html`

[6] wenn bestimmte Teile nicht benutzt werden sogar ab Java 1.1

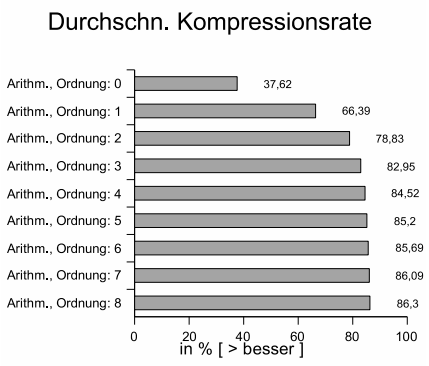

Abbildung 3.2: Durchschn. Kompressionsrate des 1. arithmetischen Verfahrens, Ordnung 0-8

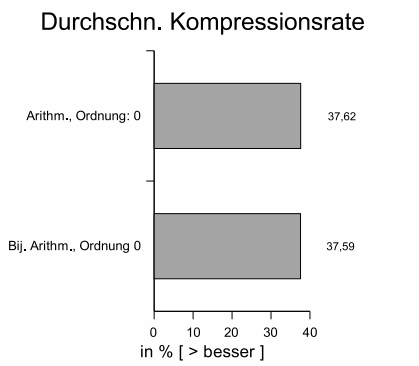

Abbildung 3.3: Durchschn. Kompressionsrate der beiden arithmetischen Verfahren, Ordnung 0

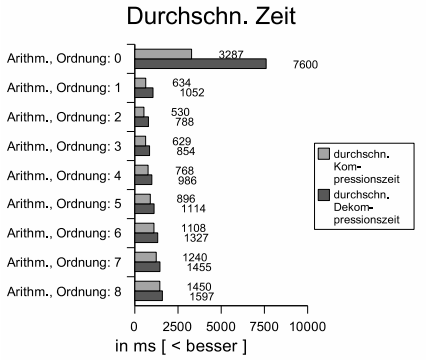

Abbildung 3.4: Durchschn. Zeitverbrauch des 1. arithmetischen Verfahrens, Ordnung 0-8

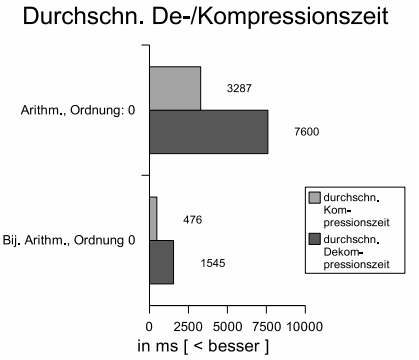

Abbildung 3.5: Durchschn. Zeitverbrauch der beiden arithmetischen Verfahren, Ordnung 0

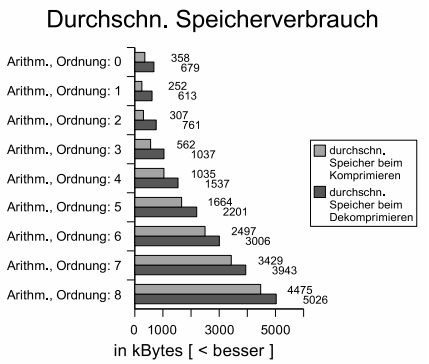

Abbildung 3.6: Durchschn. Speicherverbrauch des 1. arithmetischen Verfahrens, Ordnung 0-8

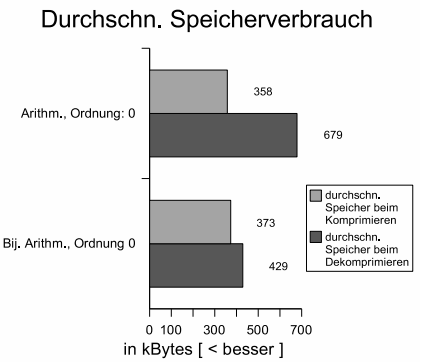

Abbildung 3.7: Durchschn. Speicherverbrauch der beiden arithmetischen Verfahren, Ordnung 0

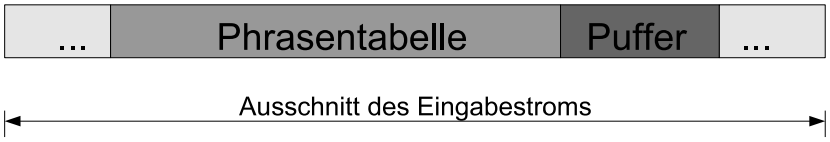

Abbildung 3.8: Prinzip des tabellengesteuerten LZ77 Verfahrens

landet in der Phrasentabelle. Wird eine Übereinstimmung erkannt wird diese durch ein Offset-Längenpaar kodiert. Eine verständliche Übersicht des Vorgangs zeigt Abbildung 3.9. Hier ist die Phrasentabelle links und der Puffer rechts dargestellt. Zwischen den untereinander aufgeführten Schritten ist die jeweilige Ausgabe zu sehen. Aus dem Beispieltext "XML und XML-Schema" wird Zeichen für Zeichen übernommen bis eine Übereinstimmung beim zweiten Auftreten von "XML" gefunden wird. An dieser Stelle wird ein Token, das aus soviel Bit besteht wie nötig sind um den Offset und die Länge beschreiben zu können, ausgegeben. Wenn z. B. die Phrasentabelle 4096 Bytes groß ist werden 12 Bit dafür benötigt, zusätzlich werden, wenn der Puffer z. B. 16 Bytes groß ist, 4 Bit benötigt um die Länge der Übereinstimmung zu kodieren. Insgesamt werden für das Kodieren des Beispieltokens also 16 Bit benötigt.

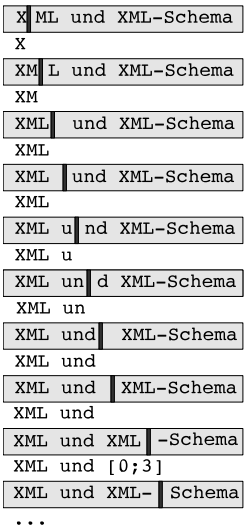

Abbildung 3.9: LZSS Kodierungsbeispiel für den Text "XML und XML-Schema"

JZlib

Die Originale Zlib Version war ursprünglich in C implementiert, sie wurde von Jean-Loup Gailly (Kompression) und Mark Adler (Dekompression) entwickelt. JZlib[7] ist ebenfalls eine LZ77 Variante mit nachgeschalteter Huffman-Kodierung

[7] *JCraft Inc.:* `http://www.jcraft.com/jzlib/`

Variante	LZSS Schwelle	max. übereinst. Länge	Grösse des LZSS Wörterbuches	Beschreibung
LH1	3	60	4096	Eine alte Standardkompressionsmethode von LHA. LZSS Kompression. Dynamische Huffman Kompression (LZSS unkomprimiert 1 Byte/LZSS Übereinstimmungslänge). Statische Huffman Kompression (obere 6 Bit der LZSS Übereinstimmungsposition)
LH2	3	256	8192	Wurde benutzt um Erweiterungen von LH1 zu testen. LZSS Kompression. Dynamische Huffman Kompression (LZSS unkomprimiert 1 Byte/LZSS Übereinstimmungslänge). Dynamische Huffman Kompression (obere 7 Bit der LZSS Übereinstimmungsposition)
LH3	3	256	8192	Wurde benutzt um Erweiterungen von LH1 zu testen. LZSS Kompression. Statische Huffman Kompression (LZSS unkomprimiert 1 Byte/LZSS Übereinstimmungslänge). Statische Huffman Kompression (obere 7 Bit der LZSS Übereinstimmungsposition)
LH5	3	256	8192	Dies ist die Standardkompressionsmethode von LHA. LZSS Kompression. Statische Huffman Kompression (LZSS unkomprimiert 1 Byte/LZSS Übereinstimmungslänge). Statische Huffman Kompression (Bit der Länge der LZSS Übereinstimmungsposition)
LZ5	3	17	4096	Alte Kompressionsmethode von LArc. Keine näheren Informationen verfügbar.
LZS	2	17	2048	Alte Kompressionsmethode von LArc. Keine näheren Informationen verfügbar.

Tabelle 3.4: Übersicht der getesteten LHA-Varianten

und wurde von JCraft Inc. unter Java Implementiert. Sie ist auch für die Mobile Java Version J2ME[8] kostenlos erhältlich. JZlib unterstützt folgende, getestete, Optionen:

- Z_DEFAULT_COMPRESSION
- Z_BEST_SPEED
- Z_BEST_COMPRESSION

Da JZlib nur eine Reimplementierung der originalen GNU Zlib Version ist, kann man hierfür die alte Dokumentation der GNU Zlib verwenden. Zlib ist in [10] und [11] beschrieben.

Zlib Komprimierte Daten setzen sich aus Blöcken, entsprechend den Blöcken die beim Einlesen der Daten vom Medium geliefert werden, zusammen. Die Blockgrößen können also beliebig groß sein. Bei nicht komprimierbaren Daten, ist die maximale Blockgröße auf 65535 Byte limitiert. Jeder Block besteht aus einer Kombination der LZ77- und der Huffman-Kodierung.

[8] J2ME: Java 2 Micro Edition - Java Laufzeitumgebung für Mobilgeräte

Alle technischen Daten über die Arbeitsweise von Zlib wurden aus [11] übernommen. Dort wird erklärt, dass Zlib Redundanzen der maximalen Länge von 258 Byte bis auf einen Abstand von 32768 Byte erkennen kann. Beim Kodieren wird nicht der Offset der Phrasentabelle, sondern der Abstand der Wiederholung beim Rückwärtslaufen im Textfenster verwendet. Für alle kodierten Werte in den komprimierten Daten, ob Literal, Abstand oder Länge wird ein Huffman-Kode verwendet. Dabei wird ein Huffman-Kodebaum für Literale und Längen, und ein anderer Huffman-Kodebaum für Abstände verwendet. Diese Bäume befinden sich immer am Anfang eines Blocks.

J2SE integrierte Verfahren Zip & Gzip

Die getestete Zip und Gzip Version ist in J2SE integriert. Gzip implementiert ebenfalls den Deflate Algorithmus und ist im Grunde nichts anderes als Zlib.Bei Zip ist es ähnlich, das ist auch nur eine Kombination von LZ77 und Huffman. Beide Verfahren sind nicht in J2ME enthalten und werden nur zum Ergebnisvergleich verwendet. Bei beiden Verfahren wurde nur die Default Option getestet.

Ergebnisse

Betrachtet man Abbildung 3.10 und Abbildung 3.11, sieht man dass die Kompressionsrate aller Verfahren relativ ähnliche Ergebnisse erzielten. Die Kompressionsrate liegt zwischen 71% und 85%, was eine recht hohe Kompressionsrate darstellt. Die beste Kompressionsrate erzielte Zlib mit der Option Z_BEST_COMPRESSION und die schlechteste erzielte LZS. In Abbildung 3.12 und Abbildung 3.13 erkennt man, dass die benötigte Zeit beim Komprimieren bei allen Verfahren deutlich höher ist, als beim Dekomprimieren. Das Komprimieren ist durchschnittlich um Faktor 3 langsamer als das Dekomprimieren. Dabei erzielte Zlib mit der Option Z_BEST_SPEED das beste Verhältnis mit 1,2 und LZS das schlechteste Verhältnis mit etwa 4. Damit liegt Zlib mit der Option Z_BEST_SPEED beim Komprimieren mit 36 ms vorne und LZS mit 131 ms hinten. Die Dekompression wird von allen Verfahren relativ schnell durchgeführt, dabei liegen die beiden J2SE integrierten Verfahren Gzip und Zip mit 24 ms vorne, dicht gefolgt von Zlib mit 25 ms für die Optionen Z_DEFAULT_COMPRESSION und Z_BEST_COMPRESSION. Die schlechteste Dekompressionszeit lieferte LH2. Betrachtet man den Speicherverbrauch in Abbildung 3.14 und Abbildung 3.15 sieht man dass hier das Verhältnis Kompression zu Dekompression genau umgekehrt ist wie beim Zeitverbrauch. Beim Dekomprimieren benötigen alle Verfahren ca. 600 kByte an Speicher was in etwa doppelt so viel Speicher wie beim Komprimieren ist. Beim Komprimieren benötigen alle Verfahren etwa 350 kByte Speicher. Lediglich die zwei J2SE integrierten Verfahren Gzip und Zip machen eine Ausnahme, diese zwei Kandidaten begnügen sich mit knapp 100 kByte.

Da die Messergebnisse relativ ähnlich sind, ist es schwer ein Verfahren zu nennen, das in allen Fällen das richtige sein kann. Da aber nur JZlib für Mobilgeräte verfügbar ist, und dieses Verfahren Optionen anbietet mit denen man die Kompressionsrate bzw. die Kompressionszeit beeinflussen kann, bietet es eine Möglichkeit, das Verfahren je nach Bedarf anzupassen. Es erzielte mit den

entsprechenden Optionen die besten Ergebnisse und deshalb ist JZlib zu bevorzugen.

3.2.3 Zusammenfassung der Ergebnisse

Nach genauer Betrachtung der Ergebnisse der statistischen und tabellengesteuerten Kompressionsverfahren sind die arithmetischen Verfahren durch ihren hohen Speicherverbrauch nicht unbedingt für Mobilgeräte zu gebrauchen. Würde man das arithmetische Verfahren der Ordnung 2 einsetzen, welches eine Speicherauslastung die in etwa den tabellengesteuerten Verfahren entspricht, würde man zwar auch eine ähnliche Kompressionsrate erzielen, die benötigte Zeit fürs Komprimieren würde aber etwa 6 mal höher sein als bei den tabellengesteuerten Verfahren, beim Dekomprimieren wäre dies sogar noch schlechter. Die tabellengesteuerten Verfahren sind recht schnell und bieten eine gute Kompressionsrate, wobei sie auch am wenigsten Speicher benötigen. Deshalb sollte beim Einsatz eines Standardkompressionsverfahrens für XML, erst recht bei ressourcenbeschränkten Mobilgeräten auf das tabellengesteuerte Verfahren zurückgegriffen werden.Nach den oben durchgeführten Messungen und Bewertungen, kommt im Rahmen dieser Diplomarbeit, nur noch JZlib in Frage. JZlib bietet, durch die unterstützten Optionen, eine gewisse Flexibilität und ist dadurch am besten geeignet.

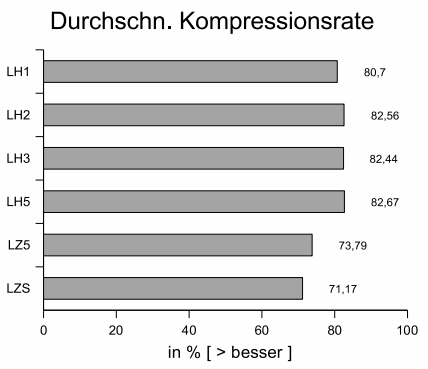

Abbildung 3.10: Durchschnittliche Kompressionsrate der LHA Verfahren

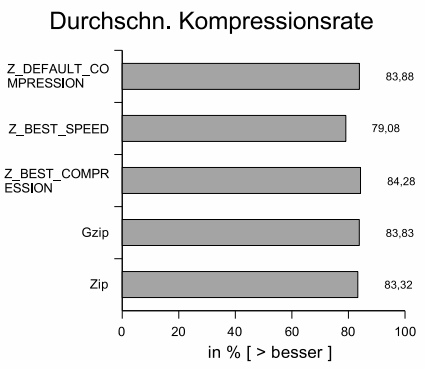

Abbildung 3.11: Durchschn. Kompressionsrate von Zlib, Gzip, Zip

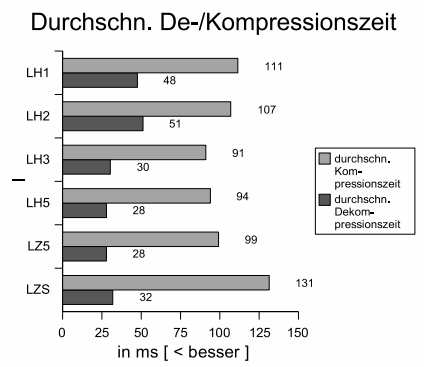

Abbildung 3.12: Durchschnittlicher Zeitverbrauch der LHA Verfahren

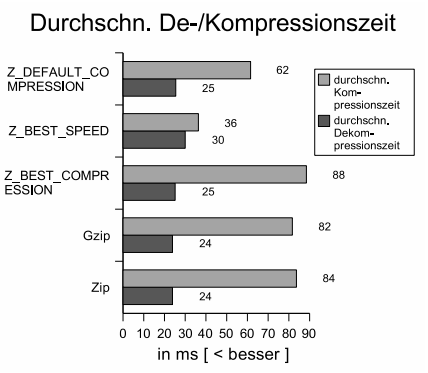

Abbildung 3.13: Durchschn. Zeitverbrauch von Zlib, Gzip, Zip

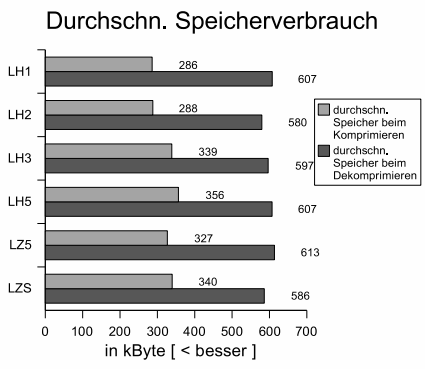

Abbildung 3.14: Durchschnittlicher Speicherbrauch der LHA Verfahren

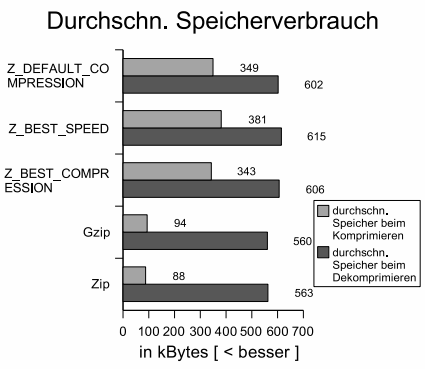

Abbildung 3.15: Durchschn. Speicherbrauch von Zlib, Gzip, Zip

Kapitel 4

XML Kompressionsverfahren

4.1 Potentielle Lösungen

Alle XML-spezifische Kompressionsverfahren die in folgenden Abschnitten vorgestellt werden sind potentielle Lösungen für die Kompression von XML-Daten und der Übertragung mit Mobilgeräten. Sie sind übrige Lösungen nach der Auswahl aus Kapitel 2.3.2 und sind in Tabelle 2.3 aufgelistet.

4.1.1 BinXML - Expway

Die BinXML Lösung von Expway scheint eine etwas weiter entwickelte Lösung zu sein. Expway bietet einen StartetKit mit einer 30-Tages Lizenz an, welcher diesen Test ermöglichte. Der BinXML StarterKit V 3.0 beinhaltet BiM, ein binäres Format zur Kodierung von XML. Laut Expway[1] ist BinXML in verschiedene Standardisierungsprozesse involviert wie z. B. ISO/MPEG-7, DVB, MPEG21, ARIB, TV-ANYTIME, 3GPP, und DAB. Dies spricht für die Lösung. Expway beschreibt in [16] dass in den MPEG-7 Tests eine durchschnittliche Kompressionsrate von 85% erreicht wurde, außerdem sei ihre Lösung unter Java etwa 4x schneller als der Xerces SAX-Parser so Expway in [14].

Durch die Unterstützung von XML-Schema ist es möglich XML-Dateien sehr kompakt zu komprimieren. Dazu verfügt Expways Lösung über einen so genannten Schemacompiler der aus einem vorliegenden XML-Schema eine vorkompilierte Schema-Datei erzeugt. Diese wird für den Decoder benötigt um die Daten wieder zu dekodieren. Alternativ können XML-Dateien auch ohne vorliegendes XML-Schema komprimiert werden. Für diesen Fall liefert Expway eine XML-Schemadatei, welche mit allen XML-Dateien die kein XML-Schema verwenden benutzt werden kann.

Die Arbeitsweise von BinXML zeigt Abbildung 4.1 welche aus [15] übernommen wurde. Der Schema Compiler erzeugt aus einer *.xsd Datei zwei Dateien.

[1] *Expway:* `http://www.expway.com`

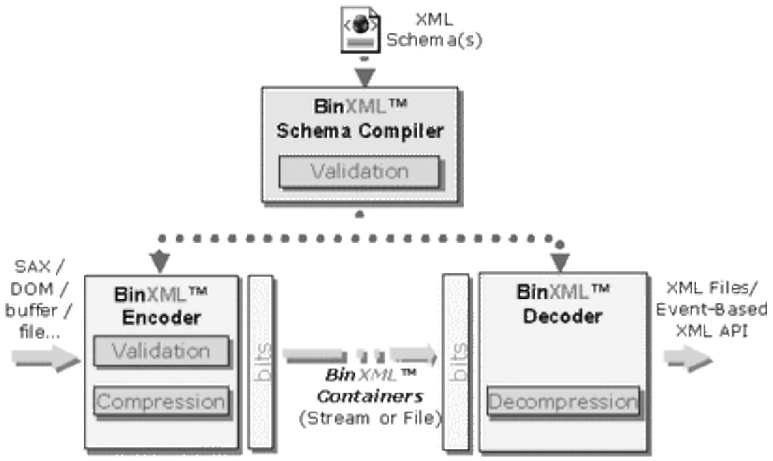

Abbildung 4.1: Arbeitsweise der BinXML Lösung von Expway

Erstens eine *.schemalight.xml Datei welche für den Komprimiervorgang benötigt wird. Diese Datei ist eine normalisierte XML-Datei im Textformat. Zweitens eine *.pbd Datei welche vom Dekomprimiervorgang benötigt wird. Sie repräsentiert das Schema in binärer und sehr kompakter Form. Zusätzlich kann gewählt werden, ob die Datei sehr kompakt, oder weniger kompakt als selbstdekodierbare Datei kodiert werden soll.

Wird die Datei sehr kompakt kodiert, ist auf Decoderseite die *.pbd Datei notwendig damit die Daten wieder Dekomprimiert werden können.

Für den Fall dass eine selfDecodable Datei erzeugt wird, ist die *.pbd Datei für das Dekomprimieren nicht erforderlich, weil die Informationen die dafür notwendig sind in der kodierten Datei enthalten sind. Darunter leidet natürlich die Kompressionsrate.

Die Tests, die hier durchgeführt wurden, beinhalten die Ausnutzung des Wissens über XML, durch die XML-Schema Datei. Dabei wurde das Verfahren mit den selfDecodable Dateien sowie den Dateien, die die *.pbd Datei zum Dekodieren benötigen, getestet.

Ergebnisse

Tabelle 4.1 zeigt die getesteten XML Dateien und deren Größe sowie die zugehörige XML-Schemadatei. Tabelle 4.2 zeigt die Größe der entsprechenden XML-Schema Dateien und deren Größe in vorkompilierter Form so wie sie zum Kodieren bzw. Dekodieren vorliegen müssen.

Zur Erläuterung der Abbildung 4.2 bis Abbildung 4.7 werden die verschiedenen Kodierungs- und Dekodierungswerkzeuge die Expway zur Verfügung stellt vorgestellt. Sie werden in den Abbildungen mit folgenden Bezeichnung gekenn-

XML-Datei	Dateigröße	XML-Schema Datei
akquise.bpel.xml	6575	BPEL4WS.xsd
service.xml	9302	wsdl.xsd
crm-gui.xml	52716	gui.xsd

Tabelle 4.1: Getestete XML-Dateien (Größenangaben in Bytes) und zugehörige XML-Schemadatei

XML-Schema Datei	Originalgröße	Schemalight Dateigröße	PBD Dateigröße
BPEL4WS	24108	57184	4967
wsdl.xsd	12163	21477	2235
gui.xsd	14721	39365	3114

Tabelle 4.2: XML-Schema Dateien und entsprechende vorkompilierte Größe der XML-Schemadatei in Bytes

zeichnet:

Kodierung:

- SelfDecodable: eine selbstdekodierbare Datei wird erzeugt
- XML-Schema: eine sehr kompakte Datei wird erzeugt (auf Decoder-Seite ist die *.pbd Datei notwendig)

Dekodierung:

- SAXDecoderSelfDecodable: der Decoder für selbstdecodierbare BinXML Dateien
- SAXLikeDecoder: ein SAX ähnlicher Decoder
- SAXDecoder: ein SAXDecoder
- BaseDecoder: laut Expway, der Decoder mit der höchsten Parsegeschwindigkeit, der kleinsten Decodergröße und dem minimalen Speicherverbrauch

In Abbildung 4.2 und Abbildung 4.3 sind die Kompressionsraten der selbstdekodierbaren Dateien und der nur mit der vorkompilierten XML-Schemadatei dekomprimierbaren Datei zu sehen. Dabei ist zu erkennen dass die selbst dekodierbaren Dateien immer eine niedrigere Kompressionsrate erzielten, und diese je nach vorliegender Datei sehr variiert. Die Kompressionsmethode, bei der die vorkompilierte XML-Schemadatei vorliegen muss, erzielt jedoch sehr hohe Kompressionsraten, selbst bei der kleinsten Datei, bei der die selbstdekodierbare Datei nur eine sehr kleine Kompressionsrate erreichte, wurde eine hohe Kompressionsrate erzielt. Bei dieser Methode wurden sogar höhere Kompressionsraten als bei den getesteten Standardkompressionsverfahren aus Kapitel 3 erzielt.

Betrachtet man die zum De-/Kodieren benötigte Zeit sieht man, dass das Dekodieren deutlich schneller ist als das Kodieren. Dieses Verfahren bietet also Vorteile für den Empfänger einer binären XML Nachricht. Das Dekodieren ist mit dem Parsen einer XML-Nachricht im Textformat gleichzusetzen. Deshalb muss ein anschließendes Binding folgen um die Daten verarbeiten zu können.

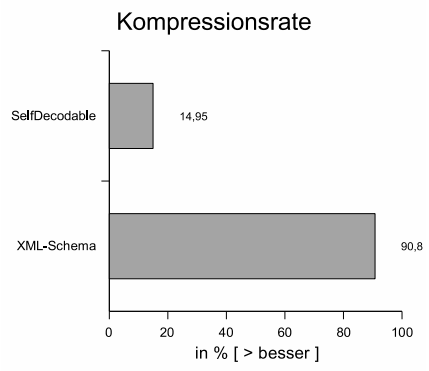

Abbildung 4.2: Durchschn. Kompressionsrate der Datei akquise.bpel.xml

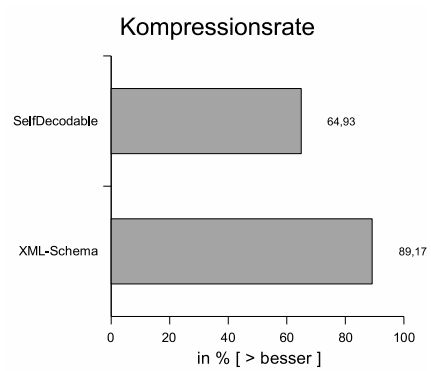

Abbildung 4.3: Durchschnittliche Kompressionsrate der Datei service.xml

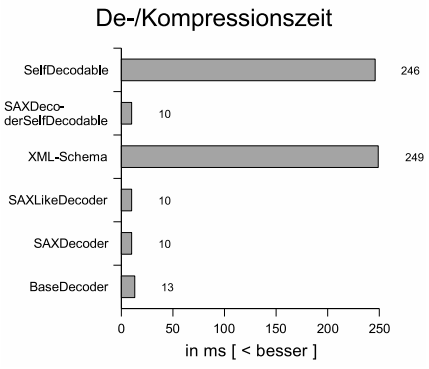

Abbildung 4.4: Durchschn. Zeitverbrauch beim De-/Komprimieren der Datei akquise.bpel.xml

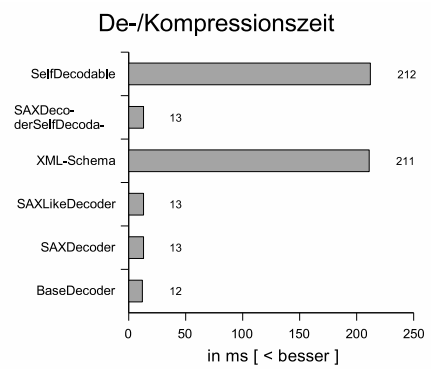

Abbildung 4.5: Durchschn. Zeitverbrauch beim De-/Komprimieren der Datei service.xml

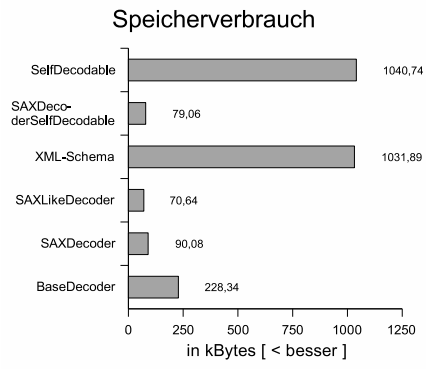

Abbildung 4.6: Durchschn. Speicherverbrauch beim De-/Komprimieren der Datei akquise.bpel.xml

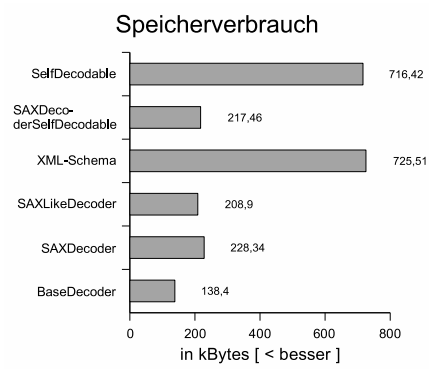

Abbildung 4.7: Durchschn. Speicherverbrauch beim De-/Komprimieren der Datei service.xml

Für die Kommunikationsseite die die Nachricht kodieren muss, macht es keinen Unterschied, ob eine selbstdekodierbare Datei erzeugt wird oder ob eine kompaktere Datei erzeugt wird, bei der der Dekodierungsprozess über eine vorkompilierte XML-Schemadatei verfügen muss. Die Verfahren sind nahezu gleich schnell. Das gleiche gilt auch für den Kodierungsprozess, hier sind auch kaum Geschwindigkeitsunterschiede zwischen den zwei getesteten Kodierungsvarianten festzustellen.

Der Speicherverbrauch während der De-/Kodierungsphase ist beim Kodieren erheblich höher als beim Dekodieren. Er ist zusätzlich von der zu komprimierenden Datei abhängig. Leider hat BinXML beim Kodieren der größten Datei einen Fehler verursacht, obwohl kein ersichtlicher Fehler vorlag. Deshalb konnte diese Datei nicht in die Messungen miteinbezogen werden. Da der Speicherverbrauch bei der kleineren Datei mit dem BaseDecoder höher ist als bei der größeren Datei, ist hier kein proportionales Verhalten erkennbar, und daher auch keine Aussage darüber möglich. Bei allen anderen getesteten Decodern ist jedoch deutlich zu erkennen, dass bei der größeren Datei der Speicherverbrauch höher ist, als bei der kleineren Datei.

BinXML erzielt höhere Kompressionsraten als die Standardkompressionsverfahren und ist beim Dekomprimieren sehr schnell. Da die Dekompression das Parsen, ohne Binding, schon beinhaltet, und schneller ist, als alle getesteten Standardkompressionsverfahren bei denen das Parsen noch folgen muss, ist dieses Verfahren für die binäre XML-Datenübertragung besonders geeignet. Der Speicherverbrauch variiert sehr und ist selbst bei der kleinsten Datei höher als 1 MByte, das ist für Mobilgeräte nicht unbedingt vertretbar. Das Seltsame ist dass, bei der größeren Datei weniger Speicher benötigt wird. Eine Aussage darüber ob der Speicherbedarf bei der Mobilversion die Expway anbietet an die Erfordernisse eines Mobilgeräts angepasst ist, kann nicht gemacht werden da Expway die Mobilversion nicht als Testversion anbietet.

4.1.2 XBIS

XBIS ist der Nachfolger des XML Stream (XMLS) Projekts aus den Jahren 2000-2001, und ist OpenSource siehe [53]. Getestet wurde Version 0.95. XBIS ist ein Kodierungsformat für XML-Dokumente das die Konvertierung von und nach XML im standard Textformat, mit Infoset Äquivalenz zwischen dem Original und dem wiederhergestellten Textformat, ermöglicht. XBIS ist somit ein Infoset-basiertes Verfahren und arbeitet mit SAX2 Event Streams. Es wurde entwickelt, um die Verarbeitungsgeschwindigkeit zu erhöhen. Dabei wurden nicht nur Geschwindigkeitsvorteile sondern auch Dateiverkleinerungen erzielt. XBIS bietet weitere Vorteile z. B. können Dokumente in normales XML im Textformat umgewandelt werden, falls sie vom Menschen gelesen werden müssen. Außerdem kann XBIS sogar Redundanzen zwischen mehreren hintereinander folgenden Dokumenten ausnutzen, wenn diese über einen Stream versendet werden.

Der Entwickler von XBIS, Dennis Sosnoski, beschreibt in [52], dass die Kompression je nach XML-Dokumententyp von 17% bis 72% reicht und dass XBIS keine Algorithmen zur Kompression von Text enthält, sondern Text als Klartext in seiner ursprünglichen Form verwendet wird. Deshalb erreicht XBIS verglichen

zu den Standardkompressionsverfahren eine relativ niedrige Kompressionsrate, allerdings bietet XBIS eine sehr hohe CPU Performanz. Außerdem erklärt Dennis Sosnoski ebenfalls in [52], dass XBIS generell etwa die 4- bis 8-fache Inputgeschwindigkeit eines SAX2 Parsers erreicht, dies wurde dort für alle XML-Dokumenttypen mit der Sun- und der IBM JVM getestet. Zudem beschreibt er dass XBIS beim Output sogar eine 7- bis 10-fach höhere Geschwindigkeit gegenüber dem SAX2 textoutput Generator erreicht.

Ergebnisse

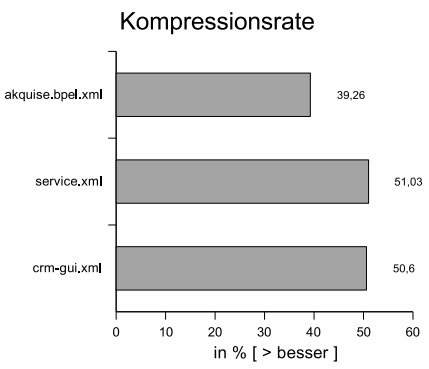

Abbildung 4.8: Durchschn. Kompressionsrate mit XBIS

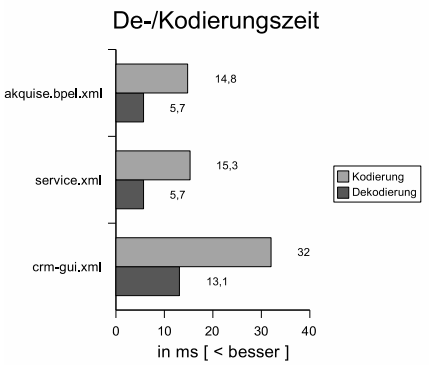

Abbildung 4.9: Durchschn. Zeitverbrauch beim De-/Komprimieren mit XBIS

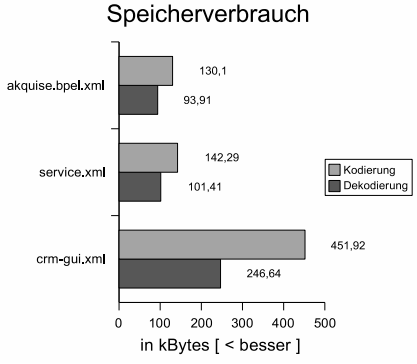

Abbildung 4.10: Durchschn. Speicherverbrauch beim De-/Komprimieren mit XBIS

Um besser vergleichbare Ergebnisse zu erreichen wurden für die Tests mit XBIS dieselben Dateien wie bei den Tests mit BinXML verwendet, siehe Tabelle 4.1. Wie erwartet ist die Kompressionsrate mit maximal 51%, siehe Abbildung 4.8 nicht besonders hoch. Allerdings zeigt Abbildung 4.9 dass XBIS beim Dekodieren etwa doppelt so schnell ist wie BinXML, und beim Kodieren ist XBIS sogar etwa 16-mal schneller als BinXML. XBIS ist beim Dekodieren schneller

als beim Kodieren, genau so wie BinXML und benötigt durchschnittlich relativ wenig Speicher. Es benötigt beim Kodieren nicht ein hohes Vielfaches des beim Dekodieren benötigten Speichers wie BinXML sondern durchschnittlich nur ca. 1,6 mal so viel Speicher wie beim Dekodieren. Der maximale Speicherverbrauch der größten Datei liegt bei etwa 452 kByte, siehe Abbildung 4.10.

XBIS ist deutlich schneller als alle Standardkompressionsverfahren und ist somit das schnellste und sparsamste Verfahren das bisher getestet wurde. XBIS hat den Vorteil dass beim Dekodieren der SAX2 Events, das Parsen überflüssig wird und somit ein Arbeitsgang gespart wird. Der einzige Nachteil ist dass XBIS wirklich rein Infoset-basiert arbeitet und kein XML-Schema unterstützt. Dies schlägt sich in der Kompressionsrate nieder wie man in Abbildung 4.8 sieht.

4.1.3 Xebu

Xebu [44] wurde an der Universität Helsinki entwickelt und ist ein Serialisierungsformat für XML. Es verwendet eine Sequenz von SAX Events. Diese können in unterschiedlichen Kodierungsformaten serialisiert werden. Leider ist Xebu nicht besonders gut dokumentiert, so dass in der verfügbaren Zeit nur die verschiedenen, unterstützten Serialisierungsformate getestet wurden und daher nur Aussagen über die Kompressionsrate gemacht werden können. Durchgeführte Tests zeigten dass nicht jedes Format kompakter als das Original ist, es wurde eine maximale Kompressionsrate von 43% erreicht. Oft wurden sogar negative Kompressionsraten erzielt, was auf eine Vergrösserung der Originaldaten zurückzuführen ist. Wie es scheint ist Xebu entweder noch in der Entwicklung oder verfolgt ein anderes Ziel als die Kompression und ist daher nicht brauchbar.

4.2 Andere Lösungen

Hier werden einige zusätzliche Lösungen die nicht als potentielle Lösungen angesehen werden können vorgestellt. Diese Lösungen sind entweder nicht bzw. noch nicht verfügbar, sind für Mobilgeräte unbrauchbar oder sind nicht in Java geschrieben, so dass sie nicht in das PI-Data-Framework eingebunden werden können. Trotzdem werden sie aus folgenden Gründen vorgestellt:

- zum Vergleich der Ergebnisse mit anderen Verfahren,
- weil die Lösung sehr bekannt oder verbreitet ist,
- weil das Verfahren sehr elegant ist oder eine gute Idee beinhaltet, die in einer eigenen Lösung eventuell auch berücksichtigt werden sollte.

4.2.1 XML Schema Tools

Es gibt eine Reihe von Werkzeugen die von OSS Nokalva[2] angeboten werden. Diese Werkzeuge sind speziell für den Einsatz von XML in einem binären Format gedacht. Dieses binäre Format ist ASN.1[3] und stammt aus der Telekommunikationsindustrie. ASN.1 ist eine sehr kompakte binäre Kodierung die seit über

[2] *OSS Nokalva:* `http://www.oss.com/`
[3] ASN.1: Abstract Syntax Notation 1

20 Jahren eingesetzt wird und nun auch in Verbindung mit XML eingesetzt werden kann. Dafür wurden mehrere Standards entwickelt. Unter anderem sind das folgende ISO/ITU-T Standards (teilweise noch nicht standardisiert):

- X.680, ASN.1 Spezifikation [23]
- X.690, BER - Basic Encoding Rules, definiert die Standardkodierungsregeln [24]
- X.691, PER - Packed Encoding Rules, definiert ein erweitertes kompakteres Kodierungsverfahren [25]
- X.693, XER - XML Encoding Rules, definiert einen Weg ASN.1 mittels XML darzustellen [26]
- X.694, definiert einen Standardisierten Weg um zwischen ASN.1 und XML zu übersetzen, es erlaubt eine Kodierung von einem Format ins andere und umgekehrt [27]

In den Tests von Objective Systems[4] [9], einem Unternehmen welches ebenfalls ASN.1 Tools vertreibt, gibt Ed Day einen zehnfachen Geschwindigkeitszuwachs bei der Benutzung von ASN.1 Kodierungstools gegenüber der gewohnten Benutzung von XML mit einem SAX-Parser an.

4.2.2 Fast Web Services

Sun Microsystems hat mit dem Ziel, eine Alternative zu XML zu schaffen, einen Prototyp namens "Fast Web Services" unter Java entwickelt, siehe [48] für mehr Informationen. Fast Web Services arbeitet mit binären Nachrichten, so dass weniger Bandbreite, weniger Speicher und eine schnellere Verarbeitung erzielt werden kann. Der Nachteil des Ganzen ist der Verlust der selbstbeschreibenden Eigenschaft von XML. Ein weiteres wichtiges Ziel bei der Entwicklung war die Austauschbarkeit zwischen XML und Fast Web Services, so dass Fast Web Services benutzt werden kann, sobald es verfügbar ist. Wenn nicht bleibt immer noch der gewohnte Weg mit XML.

Fast Web Services baut auf ASN.1 auf und benutzt diverse X.69? Standards zum Kodieren der binären Nachrichten. Durch die Benutzung von PER ist es möglich, die kompakteste und schnellste Kodierung bzw. Dekodierung im Zusammenhang mit ASN.1, zu verwenden. Dies hat den Vorteil keine neuen Technologien erfinden und einführen zu müssen. Leider ist diese Lösung noch nicht verfügbar und deshalb unbrauchbar.

Fast parallel dazu entsteht eine andere Lösung namens Fast Infoset, sie benutzt auch die ASN.1 Kodierung und wird in Fast Web Services als Ersatz dienen wenn kein XML-Schema verfügbar ist. Nähere Informationen dazu unter [49]. Fast Infoset ist schon im aktuellen Java Web Services Developer Pack Version 1.6 [56] enthalten. Leider ist diese Version erst kürzlich erschienen und deshalb konnten diesbezüglich keine Tests durchgeführt werden. Die bisherigen unter [49] präsentierten Ergebnisse von Fast Infoset sind in zwei Gruppen von

[4] *Objective Systems:* `http://www.obj-sys.com`

Dateien aufgeteilt worden, in kleine und große Dokumente. Für die kleinen Dokumente wurde beim Parsen eine Geschwindigkeitserhöhung um den Faktor 3 bis 5, und für das Generieren, hier als Serialisieren bezeichnet, eine Geschwindigkeitserhöhung um den Faktor 5 bis 10 gemessen. Die Kompressionsangaben sind durch eine Verkleinerung der Datei um den Faktor 1,3 bis 3,3 angegeben. Für die großen Dokumente wurde beim Parsen eine Geschwindigkeitserhöhung von 3 bis 4,3 und beim Serialisieren um 5 bis 11 festgestellt. Die Dateien sind um den Faktor 1,7 bis 5 verkleinert worden.

4.2.3 XMill

XMill [55] ist eine Entwicklung von AT&T und der University of Pensylvania, und ist ein Kompressionsverfahren speziell für XML. Mittlerweile existiert eine OpenSource Version von XMill, siehe [68]. XMill wurde nicht zur XML-Datenübertragung sondern zum Archivieren von XML Daten entwickelt. Durch diese Eigenschaft lässt sich XMill erstens nicht so einfach für die Datenübertragung nutzen, und zweitens ist es nicht in Java geschrieben worden.

Durch den erstgenannten negativen Aspekt lässt sich XMill in die Kategorie der Standardkompressionsverfahren eingliedern, und unterliegt den schon vorgestellten Vor- und Nachteilen eines Standardkompressionsverfahrens. Nichtsdestotrotz ist XMill sehr bekannt. Laut [54] erzielt XMill Kompressionsraten die doppelt so gut sind wie mit Gzip komprimierte XML-Dateien. Dies liegt an der speziellen Arbeitsweise von XMill, welche zuerst eine Trennung von Struktur und Daten vornimmt und anschließend die Daten gruppiert. Dies geschieht so, dass Daten mit gleicher Bedeutung zusammen in einem Block gruppiert werden. Die verschiedenen Blöcke können mit unterschiedlichen Kompressionsverfahren komprimiert werden und anschließend werden diese komprimierten Blöcke zu einer Datei zusammengefasst und nochmals mit Gzip komprimiert. Man kann sich vorstellen, dass diese aufwendige Arbeitsweise recht gute Kompressionsraten erzielt. XMill wird hier nur als Ergebnislieferant dienen, diese Ergebnisse werden dann zu Vergleichszwecken mit anderen Verfahren genutzt. Im Allgemeinen erzielt XMill ab einer Dateigröße von etwa 20 kByte bessere Kompressionsraten als Gzip, dies bestätigt Abbildung 4.11 welche aus [54] übernommen wurde.

4.3 Zusammenfassung der bisherigen Ergebnisse

Um die potenziellen Lösungen mit den anderen vorgestellten Lösungen besser vergleichen zu können, wurden zusätzlich noch weitere Tests durchgeführt. Zum einen wurde ein Parsegeschwindigkeitstest mit der Java Version des SAX Parsers Xerces 2.6.2, und zum anderen Kompressionstests mit XMill 0.9.1, Bzip2 1.0.2 und Gzip 1.3.5 durchgeführt. Die Ergebnisse der Parsegeschwindigkeit und des Speicherverbrauch des SAX Parsers zeigt Abbildung 4.12 und Abbildung 4.13 und die erzielten Kompressionsraten mit den genannten Verfahren zeigt Abbildung 4.14. Für diese Tests wurden die Dateien aus Tabelle 4.1 verwendet.

Vergleicht man die Parsezeit in Abbildung 4.12 mit der Dekodierungszeit vom XML-Schemabasierten BinXML in Abbildung 4.4 und Abbildung 4.5 ist BinXML gegenüber dem SAX Parser nur minimal schneller. BinXML benötigt

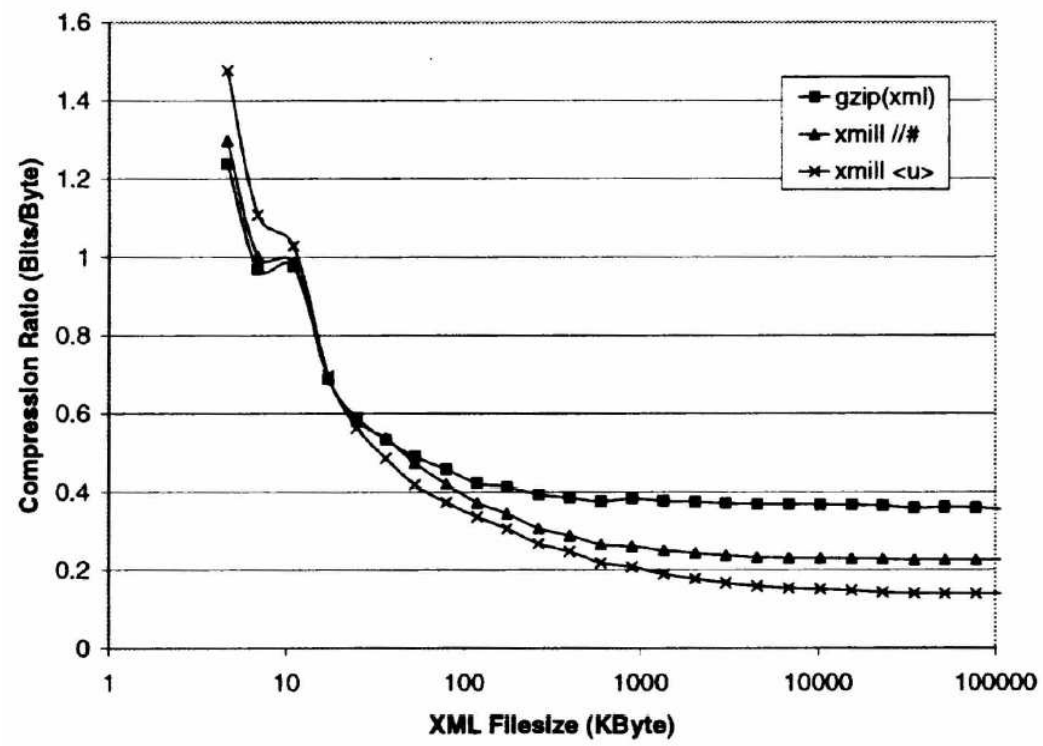

Abbildung 4.11: Kompression von XML mit XMill und Gzip

je nach eingesetztem Decoder, mal mehr Mal weniger Speicher als der SAX Parser. Der einzige nennenswerte Vorteil der BinXML bietet ist die hohe Kompressionsrate und dass dadurch weniger Bytes versendet werden müssen und somit für den Betreiber weniger Kosten entstehen. BinXML erreicht sogar höhere Kompressionsraten als alle zusätzlich getesteten Kompressionsverfahren in Abbildung 4.14. XBIS erreicht zwar die schlechtesten Kompressionsraten, ist aber das schnellste Verfahren und benötigt dabei sogar recht wenig Speicher. Durchschnittlich ist XBIS etwa doppelt so schnell wie der getestete SAX Parser. Eine Alternative, die eine höhere Kompressionsrate als XBIS erreicht und dabei schnellere Verarbeitungszeiten verspricht, ist Fast Infoset bzw. Fast Web Services.

Im Allgemeinen zeigen die bisherigen getesteten Verfahren, dass XML-Schema-basierte Verfahren neben den Standardkompressionsverfahren die höchsten Kompressionsraten erzielen, aber nicht unbedingt die schnellsten bezüglich der Verarbeitung sind. Betrachtet man die De-/Kompressionszeit, dann liegt hier das Infoset-basierte Verfahren XBIS an der Spitze, welches aber zugleich die schlechteste Kompressionsrate erzielte. Scheinbar muss man bei der Auswahl eines binären XML sich entscheiden ob man eher auf Verarbeitungsgeschwindigkeit oder auf Kompaktheit setzt.

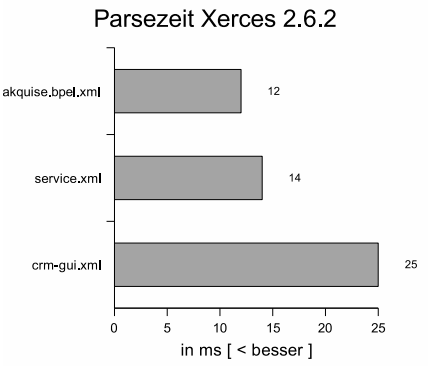

Abbildung 4.12: Xerces Parsegeschwindigkeit

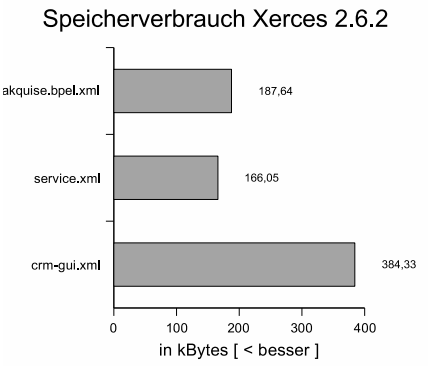

Abbildung 4.13: Xerces Speicherverbrauch

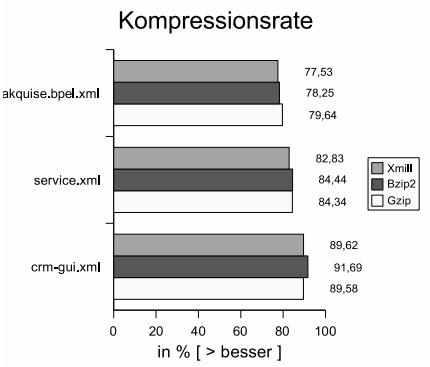

Abbildung 4.14: Kompressionsraten anderer Kompressionsverfahren (zum Vergleich) in %

4.4 Eigene Lösung

Es gibt noch keinen binären XML-Standard, der Interoperabilität gewährleistet und alle Anforderungen eines binären XML erfüllt. Es existieren zwar viele binäre XML-Lösungen, diese wurden aber meist für einen bestimmten Zweck entwickelt. Deswegen wurde eine eigene binäre Lösung entwickelt die für den mobilen Einsatz optimiert ist.

Diese eigene Lösung ist XML-Schema-basiert und arbeitet mit den PI-Data internen Datenstrukturen. Sie ist als Eratz zu dem sehr schnellen PI-Data Parser zu sehen und kann immer dann eingesetzt werden, wenn kleinere Datenmengen versendet werden sollen. Das Ziel der eigenen Lösung war es ein kompakteres binäres XML-Format zu spezifizieren. Dafür wurde ein binärer Decoder und ein buinärer Generator geschrieben.

4.4.1 Kodierungsgrundlage

Um diese Lösung zu verwirklichen wurde zuerst eine eigene Kodierung für das neue binäre XML-Format spezifiziert. Diese Kodierung beschreibt ebenfalls XML-Daten wie man sie aus der üblichen Textkodierung kennt. Der Vorteil der binären Kodierung ist dass sie sehr knapp gehalten werden kann, und dadurch einen gewissen Grad an Kompression erreicht. Die Kompression wird

- erstens durch weglassen von Elementnamen und Attributnamen welche vom XML-Schema wieder hergestellt werden können,
- zweitens durch weglassen von XML-üblichen Zeichen wie z. B. den spitzen Klammern, Leerzeichen, Eingabezeichen, Tabulatoren, Kommentar usw.

erreicht. Dafür wurden spezielle Token die die Struktur einer XML-Datei beschreiben spezifiziert. Diese Token werden den bekannten XML-Tags vorangestellt um sie zu identifizieren, und beinhalten mehrere Informationen wie z. B. um was für eine Art von XML-Tag es sich handelt (Element, Attribut...), ob das entsprechende Element Kinder besitzt und falls dies so ist wie viele usw. Wie das eigene Kodierungsformat aussieht zeigt Tabelle 4.3, Tabelle 4.4 und Tabelle 4.5.

Elemente und Attribute werden mittels Indizes identifiziert. Sie können mit Hilfe des vorangestellten Namespacepräfix der ebenfalls durch einen Index identifiziert wird aus dem XML-Schema wiederhergestellt werden. Da der Software sowieso Indizes zu den Elementen und Attributen bekannt sind, kann an dieser Stelle eine Menge Speicherplatz gespart werden. Zusätzlich zum Namespacepräfix und dem Index eines Elements wird optional die Anzahl der Kinder kodiert. Jedem Attributname folgt ein Attributwert welcher entweder durch einen Index eines Tabelleneintrags kodiert ist, falls der Attributwert schon einmal vorkam. Ansonsten wird der Attributwert als Zeichenkette mit einem vorangestellten Byte, das die Länge der Zeichenkette angibt, kodiert. Kommt ein Attribut zum ersten Mal im Eingabestrom vor, wird es in die Tabelle aufgenommen so dass ein erneutes Vorkommen des Attributwertes mit einem Index kodiert bzw. dekodiert werden kann, und somit Platz gespart wird. Nach dem Einlesen und Identifizieren eines Elements werden die Attribute und Attributwerte eingelesen. Falls das Element Kinder besitzt werden diese eingelesen bevor das Element vollständig konstruiert wird. D. h. es wird ein Tiefe-Zuerst-Algorithmus angewandt um alle Kinder eines Knotens zu ermitteln und an den Elternknoten im Konstruktor zu übergeben.

4.4.2 Kodierung & Dekodierung

Damit man ein Verständnis für das Kodierungs- und Dekodierungsverfahren erhält wird zunächst der Dekodierungsalgorithmus vorgestellt. Dabei sind alle Kodes, die aus der Kodierung entstehen können, aufgelistet und mit einem entsprechenden Verhalten des Verfahrens kommentiert. Der Dekodierungsalgorithmus verhält sich wie folgt und ist analog zum Kodierungsalgorithmus:

Bit	Wert	Bedeutung
7	1	Es folgt ein Index für den Tabelleneintrag (falls ein Element und keine Kinder, sonst folgt zuerst die Anzahl der Kinder, danach der Index für den Tabelleneintrag)
	0	Es folgt ein Längenbyte und anschliessend ein String dieser Länge (falls keine Kinder, sonst folgt zuerst die Anzahl der Kinder)
6	1	Für den Tabelleneintrag werden 2 Bytes verwendet
	0	Für den Tabelleneintrag wird 1 Byte verwendet
5-4		Siehe Tabelle für Element bzw. Attributwert
3-2		Unbenutzt
1-0	0	Namespace-Präfix
	1	Elementname
	2	Attributname
	3	Attributwert

Tabelle 4.3: Eigene binäre Kodierung der Token, Bit 7-0

Bit	Wert	Element
5	1	Element hat Kinder, es folgt die Anzahl der Kinder
	0	Element hat keine Kinder, es folgt Index für Tabelleneintrag bzw. Längenbyte
4	1	Für die Anzahl der Kinder werden 2 Bytes verwendet
	0	Für die Anzahl der Kinder wird 1 Byte verwendet

Tabelle 4.4: Eigene binäre Kodierung der Token, Bit 5-4, bei Elementen

1. nächstes Token einlesen.
   (a) 0x80: Einlesen des folgenden Byte, das den Namespacepräfix angibt.
   (b) -1: Ende des Eingabestroms erreicht.
2. nächstes Token, welches ein Element beschreibt, einlesen
   (a) 0x81: Einlesen des folgenden Byte, das den Index des Elements aus dem XML-Schema angibt; Element hat keine Kinder.
   (b) 0xC1: Einlesen der folgenden 2 Bytes, welche den Index des Elements aus dem XML-Schema angeben; Element hat keine Kinder.
   (c) 0xA1: Element hat Kinder; Einlesen des folgenden Byte, das die Anzahl der Kinder dieses Elements angibt; Einlesen des folgenden Byte, das den Index des Elements aus dem XML-Schema angibt.
   (d) 0xB1: Element hat Kinder: Einlesen der folgenden 2 Bytes, welche die Anzahl der Kinder dieses Elements angeben; Einlesen des folgenden Bytes, das den Index des Elements aus dem XML-Schema angibt.
   (e) 0xE1: Element hat Kinder: Einlesen des folgenden Byte, das die Anzahl der Kinder dieses Elements angibt; Einlesen der folgenden 2 Bytes, welche den Index des Elements aus dem XML-Schema angeben.

Bit	Wert	Attributwert
5	1	Dieser Attributwert ist ein Qname
	0	Normaler Attributwert

Tabelle 4.5: Eigene binäre Kodierung der Token, Bit 5-4, bei Attributwerten

(f) 0xF1: Element hat Kinder; Einlesen der folgenden 2 Bytes, welche die Anzahl der Kinder dieses Elements angeben; Einlesen der folgenden 2 Bytes, welche den Index des Elements aus dem XML-Schema angeben.

3. Falls Element ein SimpleType: Element bauen und an Elternelement anhängen; Weiter mit 1.

4. Falls Element ein ComplexType: bestimme die Anzahl der Attribute aus dem XML-Schema und wiederhole für jedes Attribut:

   (a) lese nächstes Token ein.

      i. 0x82: Einlesen des folgenden Byte, das den Index des Attributs aus dem XML-Schema angibt.

      ii. 0xC2: einlesen der folgenden 2 Bytes, welche den Index des Attributs aus dem XML-Schema angeben.

   (b) bestimme Attributtyp und lese nächstes Token, das den Attributwert angibt, ein.

      i. 0x03: Attributwert ist als Zeichenkette angegeben; Einlesen des folgenden Byte, das die Länge der Zeichenkette angibt; Einfügen der Zeichenkette in die Tabelle.

      ii. 0x83: Attributwert ist als Index angegeben; Einlesen des folgenden Byte, das den Index des Attributwerts aus der Tabelle angibt.

      iii. 0xC3: Attributwert ist als Index angegeben; Einlesen der folgenden 2 Bytes, welche den Index des Attributwerts aus der Tabelle angeben.

      iv. 0x23: Attributwert ist ein QName und als Zeichenkette angegeben; Einlesen des folgenden Byte das den Namespacepräfix angibt; Einlesen des folgenden Byte das die Länge der Zeichenkette angibt; Einfügen der Zeichenkette in die Tabelle.

      v. 0xA3: Attributwert ist ein QName und als Index angegeben; Einlesen des folgenden Byte, das den Namespacepräfix angibt; Einlesen des folgenden Byte das den Index des Tabelleneintrags angibt.

      vi. 0xE3: Attributwert ist ein QName und als Index angegeben; Einlesen des folgenden Byte, das den Namespacepräfix angibt; Einlesen der folgenden 2 Byte welche den Index des Tabelleneintrags angeben.

   (c) falls Element Kinder hat: gehe zu 1.

5. Bauen des Elements mit Attributen.

6. Anhängen an das Elternelement; Weiter mit 1.

Ein Beispiel einer binär kodierten Datei zeigt Abbildung 4.15. Dieser Ausschnitt einer XML-Datei wurde mit dem binären Generator erzeugt. Die Ausschnitte der original XML-Datei, siehe Abbildung 4.16 weiter unten, und der binären XML-Datei, Abbildung 4.15 beinhalten exakt dieselbe Information. Zum besseren Verständnis wurden einzelne Informationen mit Hinweisen (z. B. Header, 1, 2, 3, ...) versehen, diese Hinweise der binären Datei entsprechen den Hinweisen der Textdatei. Die originale Textdatei benötigt für den Ausschnitt 637 Bytes und die binäre Datei benötigt dafür 305 Byte. Dieser kleine Ausschnitt erreicht also schon eine Kompression von:

$$1 - \frac{305}{637} * 100 = 47,88\%$$

Erläuterung der Abbildung:

- Header: die ersten zwei Bytes identifizieren das binäre XML-Format von PI-Data, danach folgen zwei Bytes die die Version beschreiben. Das erste Byte davon beschreibt die Major-Version-Nr. und das zweite die Minor-Version-Nr. Die nächsten zwei Bytes beschreiben die Gesamtlänge der verwendeten Namespaces. Durch diese zwei Bytes ist es möglich maximal 65536 folgende Bytes für die Namespaces zu verwenden. Das ist genügend für den Anwendungsfall von PI-Data. Ab dem folgenden Byte werden die Namespaces deklariert. Dies ist wie folgt spezifiziert worden: Das erste Byte gibt den Index des Namespaces an, so wie er auch später im binären XML-Dokument vorkommt. Das zweite Byte gibt die Länge des Namespace-Strings an. Somit weiß der Decoder, wie viele Bytes er für den ersten Namespace einlesen muss. Nachdem diese Anzahl von Bytes eingelesen wurde beginnt das gleiche wieder von vorne, d. h. zuerst folgt der Index des folgenden Namespaces, danach die Anzahl der einzulesenden Bytes und schließlich der Namespace-String. Dies wird solange fortgeführt bis der Decoder die zuvor eingelesene Anzahl der Bytes, die die Gesamtlänge des Headers beschreiben, eingelesen hat. Nun folgt noch ein Byte, welches eine Länge eines Puffers der für zukünftige Zwecke reserviert wurde, beschreibt. In der momentanen Version wird diese Anzahl der Bytes und entsprechend viele folgende Bytes eingelesen und ignoriert.

Für alle folgenden Bytes gilt: das erste eingelesene Byte ist ein Token welches wie in Tabelle 4.3, Tabelle 4.4 und Tabelle 4.5 beschrieben wurde. Es gibt an um was für einen Typ von Information es sich handelt und wie die Information zu interpretieren ist (Vergleich Dekodierungsalgorithmus weiter oben).

1. Das erste Byte ist ein Token und gibt an, dass das nächste Byte ein Namespacepräfix ist. Mit diesem zweiten Byte ist es möglich, den Namespace, wie er im Header deklariert wurde, zu identifizieren. Das dritte Byte ist ein Token und gibt an dass es sich um ein Element handelt und dieses Element Kinder hat. Wie viele beschreibt das nächste Byte. Das nächste Byte ist der Index, mit welchem auf das Element mit Hilfe der vorgegebenen

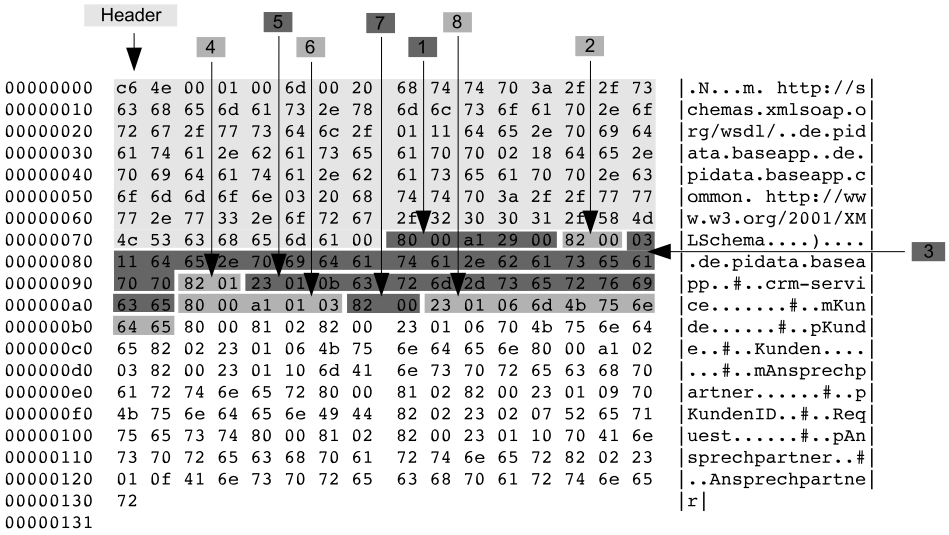

Abbildung 4.15: Ausschnitt einer binär kodierten Datei, 305 Byte (vgl. Abbildung 4.16)

Abbildung 4.16: Ausschnitt einer XML-Datei, 637 Byte (vgl. Abbildung 4.15)

Durchnummerierung durch die SchemaFactory[5] das Element identifiziert werden kann.

2. Das erste Byte ist ein Token und gibt an, dass das nächste Byte ein Attributname ist. Durch dieses zweite Byte, welches ein Index repräsentiert, ist es möglich, auf den Attributnamen zurückzuschließen.

3. Das erste Byte ist ein Token und gibt an, dass ein Attributwert folgt und dass dieser Attributwert als Bytefolge bzw. String kodiert ist. Das zweite Byte gibt die Länge des einzulesenden Strings an. Die folgenden Bytes ergeben den Attributwert.

4. Siehe 2.

5. Das erste Byte ist ein Token und gibt an, dass ein Attributwert folgt und dass dieser Attributwert ein Qualified Name ist und als Bytefolge bzw. String kodiert ist. Das zweite Byte gibt den Namespacepräfix des Qualified Name an und das dritte Byte gibt die Länge des einzulesenden Strings an. Die folgenden Bytes ergeben den Rest des Qualified Name.

6. Siehe 1.

7. Siehe 2.

8. Siehe 5.

9. usw.

4.4.3 Messung und Bewertung

Alle folgenden Abbildungen in diesem Abschnitt, bis auf Abbildung 4.17, zeigen einen Vergleich der eigenen binären Lösung gegenüber dem PI-Data Parser für XML-Dokumente im Textformat. Dies wurde zum Vergleich der Geschwindigkeit und dem Speicherbedarf gewählt, weil der PI-Data Parser und Generator speziell für Mobilgeräte entwickelt ist und somit für diesen Einsatz schon optimiert ist. Verglichen zu einem SAX Parser, siehe Abbildung 4.12, ist der PI-Data Parser, der das Binding schon enthält, trotzdem fast 3-mal schneller.

Abbildung 4.17 zeigt die erzielten Kompressionsraten der getesteten Dateien aus Tabelle 4.1. Es fällt auf, dass das eigene Verfahren eine ziemlich konstante Kompressionsrate erzielt. Bei allen anderen XML-spezifischen Kompressionsverfahren nahm die Kompressionsrate bei der kleinsten Datei gegenüber den anderen beiden Dateien ab. Diese relativ konstante Kompressionsrate ist durch die Eigenschaft der eigenen Kodierung gegeben. Diese lässt Elementnamen und Attributnamen weg und kodiert nur Attributwerte als Strings. Betrachtet man dies sehr grob muss nur etwa ein Drittel der hauptsächlichen Information in einem XML-Dokument ausführlich kodiert werden, dies bestätigt die Kompressionsrate mit knappen 66%. Die Kompressionsrate ist zwar nicht sehr hoch, betrachtet man aber den Zeitverbrauch in Abbildung 4.18 und Abbildung 4.19 sieht man, dass die binäre De-/Kodierung verglichen zu Expways BinXML (Abbildung 4.4 und Abbildung 4.5) deutlich schneller ist. Die Dekodierung des eigenen Verfahrens ist etwa 3-mal so schnell und die Kodierung ist durchschnittlich sogar etwa

[5] Diese Quellcode-Datei wird automatisch von einem Teil der Software aus dem vorliegenden XML-Schema generiert, dabei werden Elemente und Attribute mit Indizes versehen.

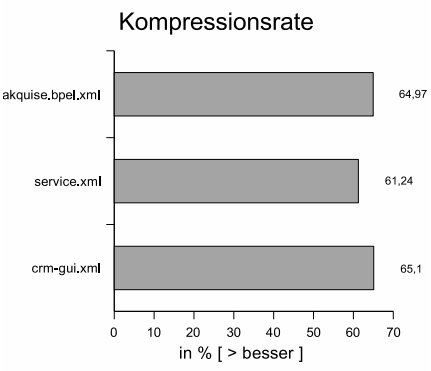

Abbildung 4.17: Kompressionsrate der eigenen binären Lösung

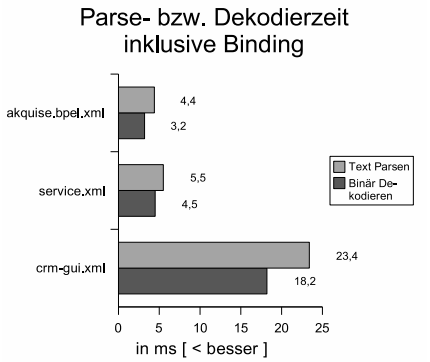

Abbildung 4.18: Durchschn. Zeitverbrauch beim Dekomprimieren

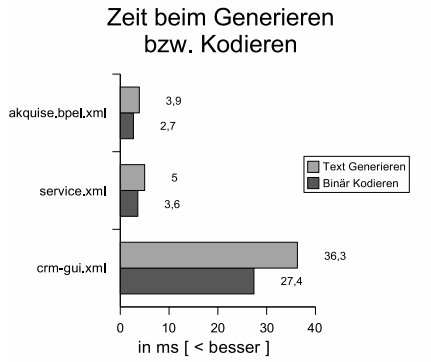

Abbildung 4.19: Durchschn. Zeitverbrauch beim Komprimieren

75-mal schneller. Dabei ist zu Bedenken, dass das eigene Verfahren zusätzlich zum Dekodieren ein Binding durchführt, welches die gemessene Zeit enthält, was bei allen anderen getesteten Verfahren nicht der Fall ist. Bis auf das Dekodieren der größten Datei, ist das eigene binäre Verfahren sogar in allen Fällen um ein paar wenige ms schneller als XBIS.

Der Speicherverbrauch ist in den meisten Fällen geringer als beim PI-Data Parser. Ausnahmen gibt es beim Kodieren der größten Datei, hier werden etwa 180 kByte mehr benötigt, und beim Dekodieren der mittelgroßen Datei, hier werden nur etwa 3 kByte mehr benötigt. Verglichen zum SAX Parser benötigt das eigene Verfahren für kleinere Dateien bis zu 4-mal weniger Speicher. Erst bei der größten Datei sind der SAX Parser und die eigene Dekodierung gleich auf. Vergleicht man den Speicherverbrauch mit BinXML benötigt das eigene Verfahren etwa 13 bis 20 mal weniger Speicher beim Kodieren, beim Dekodieren benötigt es bei der kleinsten Datei nur minimal weniger Speicher und für die mittelgroße Datei etwa halb so viel Speicher wie der sparsamste BinXML Decoder.

Verglichen mit XBIS hat das eigene binäre Verfahren nur beim Dekodie-

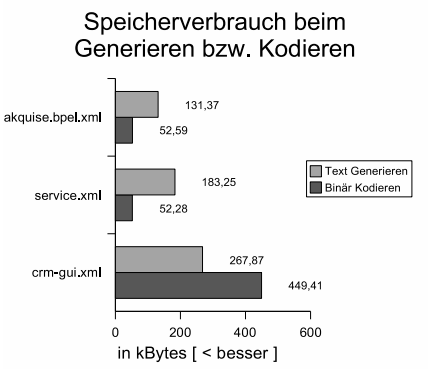

Abbildung 4.20: Durchschn. Speicherverbrauch beim Dekomprimieren

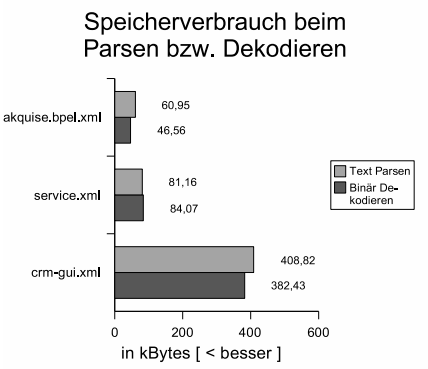

Abbildung 4.21: Durchschn. Speicherverbrauch beim Komprimieren mit der eigenen binären Lösung

ren der größten Datei einen Nachteil. Nur in diesem Fall benötigt es ca. 140 kByte mehr Speicher und ist auch etwa um 5 ms langsamer. Trotzdem bietet das eigene Verfahren hier einen Vorteil: Es führt das Binding während des Dekodierens aus, was bei XBIS nicht der Fall ist, und ist dadurch im Allgemeinen doch wieder schneller als XBIS. Die etwas höhere Kompressionsrate des eigenen binären Verfahrens, und die allgemein schnellere Verarbeitungsgeschwindigkeit zeigt, dass das eigene binäre Verfahren, für den Anwendungsfall von PI-Data, besser ist, als alle bisher getesteten XML-spezifischen Kompressionsverfahren und Standardkompressionsverfahren. Es ist das sparsamste Verfahren bezüglich des Speicherverbrauchs, das schnellste Verfahren beim De-/Kodieren, und hat eine akzeptable Kompressionsrate.

Die Kompressionsrate mit knappen 66% könnte sogar noch erhöht werden wenn man bedenkt dass es nur 16 verschiedene Token gibt, momentan dafür aber 8 Bit verwendet werden. Würde man für ein Token nur 4 Bit verwenden, was bei 16 verschiedenen Möglichkeiten ausreicht, könnte man für jedes Token 50% der Bit einsparen. Die eigene binäre Lösung bietet sogar noch mehr Potential zur Kompression. Man könnte z. B. alle Strings mit einem Standardkompressionsverfahren noch kompakter darstellen.

Kapitel 5

Fazit

In dieser Diplomarbeit wurde nach einem Kompressionsverfahren für XML gesucht, welches Speicherbedarf und Transfervolumen von XML im Textformat reduziert, und dabei nicht nur ressourcensparender sondern auch performanter arbeitet. Da es noch keinen binären XML-Standard gibt, wurden Standardkompressionsverfahren und vorhandene binäre XML-Lösungen untersucht.

Die Untersuchung der Standardkompressionsverfahren wie z. B. Gzip ergab, dass Standardkompressionsverfahren zwar eine hohe Kompressionsrate erzielen, sie aber hohe Ansprüche an die schwach ausgestatteten Mobilgeräte stellen. Sie sind deshalb nur in Sonderfällen von Nutzen.

Die vorhandenen binären XML-Lösungen bieten im Allgemeinen zwar Vorteile gegenüber den Standardkompressionsverfahren, allerdings haben die bisherigen binären XML-Lösungen einen negativen Aspekt: Die Ergebnisse zeigten, dass eine binäre Lösung entweder eine hohe Performanz oder eine hohe Kompression bietet. Keine der getesteten binären XML-Lösungen bot die wünschenswerte Kombination aus hoher Performanz und Kompressionsrate.

Deshalb wurde ein eigenes binäres Verfahren entwickelt, welches einen Kompromiss bezüglich Performanz und Kompression eingeht. Diese eigene Lösung erwies sich als die schnellste aller vorgestellten und getesteten Standardkompressionsverfahren und binären XML-Lösungen. Sie bietet zudem eine akzeptable Kompressionsrate von ca. 66% und benötigt gegenüber den getesteten Verfahren weniger Speicher. Um trotzdem weiterhin mit fremden Systemen kommunizieren zu können, wurde die eigene binäre Lösung so entwickelt, dass sie gegen den PI-Data Parser austauschbar ist. Somit bietet die eigene binäre Lösung alle wünschenswerten Eigenschaften eines binären XML und ist für den Einsatz im PI-Data Framework bestens geeignet.

5.1 Ausblick

Die Entwicklung eines binären XML-Standards ist sehr aufwendig. Dabei spielen viele Dinge eine Rolle. Die wichtigsten Dinge, die einem binären XML-Standard im Wege stehen sind jedoch die vielen Use-Cases bei denen XML zum Einsatz kommt und die dabei auftretenden Anforderungen. Die Anforderungen sind so unterschiedlich, dass es eine grosse Herausforderung für das W3C darstellt, einen binären XML-Standard in einer angemessenen Zeit zu spezifizieren. Wie lange es noch dauern wird, ist nicht vorherzusagen. Nach persönlicher Einschätzung, könnte dies durchaus noch einige Jahre dauern. Trotz allem gibt es inzwischen eine Menge Entwicklungen bezüglich eines binären XML. Kommt man um eine schnellere Verarbeitung bzw. eine kompakte Darstellung von XML nicht herum, kann man auf vorhandene Lösungen zurückgreifen oder selbst eine Entwickeln.

Aus persönlicher Sicht sind Fast Web Services und Fast Infoset zwei würdige und vielversprechende Kandidaten für einen binären Standard, da sie beide die kompakte und bewährte ASN.1 Kodierung benutzen. Erste Ergebnisse, siehe [48] und [49], zeigen überzeugende Werte. Ob diese Lösungen allerdings allen Anforderungen des W3C genügen und einen Standard bereitstellen können bleibt abzuwarten.

Literaturverzeichnis

[1] *Robin Berjon:* Expway's Position Paper on Binary Infosets (07.08.2003) http://www.w3.org/2003/08/binary-interchange-workshop/15-Expway-PositionPaper-20031020.zip (17.03.2005)

[2] *Dr. Craig S. Bruce:* CubeWerx Position Paper for Binary Interchange of XML (2003) http://www.w3.org/2003/08/binary-interchange-workshop/05-cubewerx-position-w3c-bxml.pdf (17.03.2005)

[3] *Don Brutzman, Don McGregor, Alan Hudson:* XML Binary Serialization using Cross-Format Schema Protocol (XFSP) and XML Compression Considerations for Extensible 3D (X3D) Graphics (28.09.2003) http://www.w3.org/2003/08/binary-interchange-workshop/40-BrutzmanXmlBinarySerializationUsingXfspW3cWorkshopSeptember2003.pdf (17.03.2005)

[4] *James Cheney:* Compressing XML with Multiplexed Hierarchical PPM Models (24.11.2000) http://www.cs.cornell.edu/People/jcheney/xmlppm/paper/paper.html (09.04.2005)

[5] *Michael Cokus, Daniel Winkowsky:* XML Sizing and Compression Study For Military Wireless Data (12/2002) http://www.idealliance.org/papers/xml02/dx_xml02/papers/06-02-04/06-02-04.pdf (17.03.2005)

[6] Mike Cokus, Scott Renner, Dan Winkowski, The Need for Standard Schema-based and Hybrid Compression http://www.w3.org/2003/08/binary-interchange-workshop/25-MITRE-USAF-Binary-XML.htm (17.03.2005)

[7] *Mike Conner:* CBXML: Experience with Binary XML (08.08.2003) http://www.w3.org/2003/08/binary-interchange-workshop/19-IBM-CBXML-W3C-Submission-updated.zip (17.03.2005)

[8] *Nigel Dallard:* The Use of Binary Representations of XML Information Sets in Digital Broadcasting Systems http://www.w3.org/2003/08/binary-interchange-workshop/06-NDS-Position-Paper.pdf (17.03.2005)

[9] *Ed Day:* The Use of ASN.1 Encoding Rules for Binary XML http://www.obj-sys.com/docs/ASN1forBinXML.pdf (23.05.2005)

[10] *P. Deutsch, J-L. Gailly:* ZLIB Compressed Data Format Specification version 3.3 (05/1996)
`ftp://ftp.uu.net/pub/archiving/zip/doc/rfc1950.txt` (28.07.2005)

[11] *P. Deutsch:* DEFLATE Compressed Data Format Specification version 1.3 (05/1966)
`ftp://ftp.uu.net/pub/archiving/zip/doc/rfc1951.txt` (28.07.2005)

[12] *Leigh Dodds:* Good Things Come In Small Packages (22.03.2000)
`http://www.xml.com/pub/a/2000/03/22/deviant/index.html` (09.04.2005)

[13] *Bill Eller:* Binary Interchange of XML Information (11.08.2003)
`http://www.w3.org/2003/08/binary-interchange-workshop/23a-L3IS_BinaryXML_Position_11Aug03.pdf` (17.03.2005)

[14] *Expway:* Benchmarks (2003)
`http://www.expway.com/telechargement/1112626524.pdf` (02.08.2005)

[15] *Expway:* Features
`http://www.expway.com/binxml.php` (04.08.2005)

[16] *Expway:* Produktbeschreibung
`http://www.expway.com/bim.php` (02.08.2005)

[17] *Martin Fiedler:* Projektarbeit Datenkompression (2000)
`http://www-user.tu-chemnitz.de/~mfie/compproj/index.htm` (25.03.2005)

[18] *Marc Girardot, Neel Sundaresan:* Millau: an encoding format for efficient representation and exchange of XML over the Web
`http://www9.org/w9cdrom/154/154.html` (17.03.2005)

[19] *Oliver Goldman, Larry Masinter:* Position on the Binary Interchange of XML Infosets
`http://www.w3.org/2003/08/binary-interchange-workshop/presentations-adobe.pdf` (17.03.2005)

[20] *Jörg Heuer, Andreas Hutter:* Reply to Call for Participation in W3C Workshop on Binary Interchange of XML Information Item Sets (08.08.2003)
`http://www.w3.org/2003/08/binary-interchange-workshop/39-siemens-Brief_W3C_Workshop_030809_1.pdf` (17.03.2005)

[21] *Christian Horn:* Binary XML Transfer using Direct Compilation Techniques (11.08.2003)
`http://www.w3.org/2003/08/binary-interchange-workshop/32-OSS-Nokalva-Position-Paper-updated.pdf` (17.03.2005)

[22] *Intelligent Compression Technologies, Inc.:* XML-Xpress
`http://www.ictcompress.com/products_xmlxpress.html` (14.04.2005)

[23] *International Telecommunication Union:* ITU-T X.680: Abstract Syntax Notation One (ASN.1): Specification of basic notation (07/2002)
`http://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf` (09.08.2005)

[24] *International Telecommunication Union:* ITU-T X.690: ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Rules (DER), (07/2002) `http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf` (09.08.2005)

[25] *International Telecommunication Union:* ITU-T X.691: ASN.1 encoding rules: Specification of Packed Encoding Rules (PER), (07/2002) `http://www.itu.int/ITU-T/studygroups/com17/languages/X.691-0207.pdf` (09.08.2005)

[26] *International Telecommunication Union:* ITU-T X.693: XER encoding instructions and EXTENDED-XER (10/2003) `http://www.itu.int/ITU-T/studygroups/com17/languages/X693amd1.pdf` (09.08.2005)

[27] *International Telecommunication Union:* ITU-T X.694: ASN.1 encoding rules: mapping W3C XML schema definitions into ASN.1 (01/2004) `http://www.itu.int/ITU-T/studygroups/com17/languages/X694.pdf` (09.08.2005)

[28] *Michel Ishizuka:* Informationen zu den unterstützten LHA Varianten `http://homepage1.nifty.com/dangan/en/Content/Program/Java/jLHA/Notes/CompressMethod.html` (28.07.2005)

[29] *Jaakko Kangasharju, Kimmo Raatikainen:* Byte-efficient Representation of XML Messages `http://www.w3.org/2003/08/binary-interchange-workshop/08-xebu.pdf` (17.03.2005)

[30] *Eric Lemoine, Michael Leventhal:* XML Binary Infosets: Position Paper from Tarari (28.09.2003) `http://www.w3.org/2003/08/binary-interchange-workshop/27-tarari-BinaryInterchangeOfXML.pdf` (17.03.2005)

[31] *Dmitry Lenkov:* Binary XML `http://www.w3.org/2003/08/binary-interchange-workshop/31-oracle-BinaryXML_pos.htm` (17.03.2005)

[32] *Steve Lewontin:* Nokia Position Paper: W3C Workshop on Binary Interchange of XML Information Item Sets `http://www.w3.org/2003/08/binary-interchange-workshop/02-Nokia-Position-Paper_02.htm` (17.03.2005)

[33] *Rick Marshall:* Binary Represantation of XML - A Position (11.08.2003) `http://www.w3.org/2003/08/binary-interchange-workshop/13-XML-Binary-Representation.ps` (17.03.2005)

[34] *Kazunori Matsumoto, Arei Kobayashi, Naomi Inoue:* Implementation and Evaluation of a Binary Interchange System for XML-Applications in a Cellar Phone `http://www.w3.org/2003/08/binary-interchange-workshop/34-KDDI-Binary-XML.pdf` (17.03.2005)

[35] *Vance McCarthy:* Scaling XML to High-Volume – Dos and Don'ts (17.03.2005)
http://www2003.org/cdrom/papers/poster/p059/WWW2003_Poster_59.html (10.04.2005)

[36] *David Mertz:* XML Matters: XML and compression (01.09.2001)
http://www-106.ibm.com/developerworks/xml/library/x-matters13.html (11.08.2005)

[37] *Mark Nelson:* Datenkomprimierung: Effiziente Algorithmen in C. Hannover: Verlag Heinz Heise 1993

[38] *Uche Ogbuji:* XML and Compression (04.07.2000)
http://xml.coverpages.org/xmlAndCompression.html (17.03.2005)

[39] *David Orchard, Mark Nottingham:* On XML Optimization (2003)
http://www.w3.org/2003/08/binary-interchange-workshop/26-bea-BinaryXMLWS.pdf (17.03.2005)

[40] *OSS Nokalva Inc.:* Alternative binary representations of the XML Information Set based on ASN.1 (11.08.2003)
http://www.w3.org/2003/08/binary-interchange-workshop/32-OSS-Nokalva-Position-Paper-updated.pdf (17.03.2005)

[41] *Shankar Pal, Jonathan Marsh, Andrew Layman:* A Case against Standardizing Binary Representation of XML
http://www.w3.org/2003/08/binary-interchange-workshop/29-MicrosoftPosition.htm (17.03.2005)

[42] *Hemil Patel, Derek Lau, Deepak Kulkarni:* Compressing Aviation Data in XML Format (2003)
http://ase.arc.nasa.gov/publications/pdf/2003_01_3011.pdf (17.03.2005)

[43] *Santiago Pericas-Geertsen:* W3C Workshop on Binary Interchange of XML Infosets (01.02.2005)
http://www.idealliance.org/papers/dx_xml03/papers/05-01-02/05-01-02.pdf (14.05.2005)

[44] *Kimmo Raatikainen:* Fuego Core Projekt (27.04.2005)
http://www.hiit.fi/fuego/fc/index.html (08.08.2005)

[45] *Michael Rys, Shankar Pal, Jonathan Marsh, Andrew Layman:* Standardize Binary Representation of XML? (2003)
http://www.w3.org/2003/08/binary-interchange-workshop/presentations-microsoft.pdf (17.03.2005)

[46] *Atsushi Sakakibara:* Serializing DOM Method
http://www.w3.org/2003/08/binary-interchange-workshop/21-PositionPaper_MediaFusion.zip (17.03.2005)

[47] *Paul Sandoz, Santiago Pericas-Geertsen, Kohsuke Kawaguchi, Marc Hadley:* Fast Web Services
http://www.w3.org/2003/08/binary-interchange-workshop/01-FWS_Sun.pdf (17.03.2005)

[48] *Paul Sandoz, Santiago Pericas-Geertsen, Kohuske Kawaguchi, Marc Hadley, Eduardo Pelegri-Llopart:* Fast Web Services (08/2003)
`http://java.sun.com/developer/technicalArticles/WebServices/fastWS/` (02.08.2005)

[49] *Paul Sandoz, Alessando Triglia, Santiago Pericas-Geertsen:* Fast Infoset (06/2004)
`http://java.sun.com/developer/technicalArticles/xml/fastinfoset` (02.08.2005)

[50] *Ronald Schmelzer:* Will Binary XML Solve XML Performance Woes? (16.11.2004)
`http://www.zapthink.com/report.html?id=ZAPFLASH-11162004` (17.03.2005)

[51] *John C. Schneider:* Theory, Benefits and Requirements for Efficient Encoding of XML Documents (2003)
`http://www.w3.org/2003/08/binary-interchange-workshop/30-agiledelta-Efficient-updated.html` (17.03.2005)

[52] *Dennis Sosnoski:* Position Paper W3C Workshop on Binary Interchange of Infoset Items
`http://www.w3.org/2003/08/binary-interchange-workshop/09-Sosnoski-position-paper.pdf` (17.03.2005)

[53] *Dennis Sosnoski:* XBIS
`http://www.xbis.org` (08.08.2005)

[54] *Werner Streitberger:* XML Kompression (26.07.2001)
`http://wwwbayer.in.tum.de/lehre/SS2001/HSEM-bayer/ausarbeitung12.pdf` (03.03.2005)

[55] *Dan Suciu:* XMill
`http://www.cs.washington.edu/homes/suciu/XMILL/` (29.07.2005)

[56] *Sun Microsystems:* Java Web Services Developer Pack
`http://java.sun.com/webservices/jwsdp/index.jsp` (02.08.2005)

[57] *Systematic Software Engineering Ltd.:* The W3C Workshop on Binary Interchange of XML Information Item Sets: Position Paper Systematic Software Engineering (11.08.2003)
`http://www.w3.org/2003/08/binary-interchange-workshop/22-SSE-0001W3CPositionPaper.pdf` (17.03.2005)

[58] *W3C:* XML Binary Characterization Working Group Public Page
`http://www.w3.org/XML/Binary/` (17.03.2005)

[59] *W3C:* XML Binary Characterization
`http://www.w3.org/TR/xbc-characterization/` (17.03.2005)

[60] *W3C:* XML Binary Characterization Measurement Methodologies
`http://www.w3.org/TR/xbc-measurement/` (17.03.2005)

[61] *W3C:* XML Binary Characterization Properties
`http://www.w3.org/TR/xbc-properties/` (17.03.2005)

[62] *W3C:* Report From the W3C Workshop on Binary Interchange of XML Information Item Sets
`http://www.w3.org/2003/08/binary-interchange-workshop/Report.html` (17.03.2005)

[63] *W3C:* XML Binary Characterization Use Cases
`http://www.w3.org/TR/xbc-use-cases/` (17.03.2005)

[64] *Stephen D. Williams:* Position Paper for "The W3C Workshop on Binary Interchange of XML Information Item Sets"(08.08.2003)
`http://www.w3.org/2003/08/binary-interchange-workshop/10-w3cbisposition_sdw.html` (17.03.2005)

[65] *Jimmy Zhang, Kevin Lovette:* XimpleWare W3C Position Paper
`http://www.w3.org/2003/08/binary-interchange-workshop/20-ximpleware-positionpaper-updated.htm` (17.03.2005)

[66] Artikelsammlung zum Thema: XML and Compression (04.07.2000)
`http://xml.coverpages.org/xmlAndCompression.html` (09.04.2005)

[67] Compression FAQ
`http://www.faqs.org/faqs/compression-faq/` (11.04.2005)

[68] OpenSource Version von XMill
`http://sourceforge.net/projects/xmill` (29.07.2005)