HUFFMAN CODING BASED JAVASCRIPT COMPRESSSION FOR AJAX APPLICATIONS

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Sneh Kiranbhai Limbachia
Summer 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Sneh Kiranbhai Limbachia:

Huffman Coding Based JavaScript Compression for AJAX Applications

[Signature]
Carl Eckberg, Chair
Department of Computer Science

[Signature]
Subrata Bhattacharjee
Department of Mechanical Engineering

[Signature]
Christopher Paul Paolini
College of Engineering

06/24/2011
Approval Date
Copyright © 2011

by

Sneh Kiranbhai Limbachia

All Rights Reserved
DEDICATION

This thesis is dedicated to my parents, friends and my wife.
Huffman coding is an algorithm used for lossless data compression. The Huffman scheme uses a table of frequency of occurrence for each symbol (or character) in the input stream. Each character of input is encoded into a variable-length bit string and used for constructing a Huffman Tree. In this work, Huffman coding is used to compress JavaScript within an AJAX framework. On the Web application server, JavaScript source code is Huffman encoded by classification and stored in a relational database in Base64 format. When a particular script or a set of scripts are requested by a client browser, the application server returns the Base64 representation of the encoded script within a SOAP message. The browser, having first loaded a minimal uncompressed decoding script, transforms, decodes, and dynamically loads the received JavaScript payload. The performance of this technique is measured and compared against other contemporary JavaScript minifiers and compression systems such as Dojo ShrinkSafe, Douglas Crockford’s JSMin, Dean Edwards’ Packer, and the Yahoo! YUI Compressor. Performance metrics are captured and analyzed by loading an example Rich Internet Application (RIA) over networks at different data rates, both with and without the use of a minifier or compressor. The ClassTA Course Management System developed at San Diego State University is used as a baseline RIA for gathering performance metrics. ClassTA is an AJAX-based Web application that features the look and feel of a desktop application and consists of tens of thousands of lines of JavaScript. Conclusions are drawn from the results of employing compressors over different data rates using different browsers with respect to loading and application startup time.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
<th>ABSTRACT</th>
<th>LIST OF TABLES</th>
<th>LIST OF FIGURES</th>
<th>ACKNOWLEDGEMENTS</th>
<th>CHAPTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td></td>
<td>ix</td>
<td>x</td>
<td>xii</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>TECHNOLOGIES AND PREVIOUS RESEARCH WORK</td>
<td>1.1 Web 2.0</td>
<td>1.2 JavaScript</td>
<td>1.3 JSON Framework</td>
<td>1.4 AJAX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.5 Technique for High Performance Data Compression</td>
<td>1.5.1 Types of Redundancy</td>
<td>1.5.2 Various Methods of Data Compression</td>
<td>1.6 Base64 Encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.5.1.1 Character Distribution</td>
<td>1.5.2.1 Huffman Encoding</td>
<td>1.6.1 YUI Compressor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.5.1.2 Character Repetition</td>
<td>1.5.2.2 Run Length Encoding</td>
<td>1.6.2 JSMIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.5.1.3 High Usage Patterns</td>
<td>1.5.2.3 Programmed Compression</td>
<td>1.6.3 Packer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.5.1.4 Positional Redundancy</td>
<td>1.5.2.4 Adaptive Compression</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.7 Web Workers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.8 JavaScript Compression Tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.8.1 YUI Compressor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.8.2 JSMIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.8.3 Packer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.8.4 Shrink Safe...............................................................15
1.8.5 Comparison.............................................................16

2 HUFFMAN ENCODING......................................................17
2.1 Data Compression Encoding Type..............................18
2.1.1 Lossless Compression Techniques.........................18
2.1.2 Lossy Data Compression.........................................20
2.1.3 Lossless vs. Lossy Compression.............................20
2.2 Fixed Length vs. Variable Length Coding...................21
2.3 Huffman Algorithm...................................................22
2.4 Decoding Huffman Codes..........................................26

3 INTRODUCTION TO CLASS TA.........................................29
3.1 Background..............................................................30
3.2 Use of Huffman Encoding in ClassTA.........................32

4 SYSTEM DESIGN AND IMPLEMENTATION.....................33
4.1 Class Diagram..........................................................33
4.2 Use Case Diagram.....................................................36
4.3 Sequence Diagram....................................................36
4.4 Data Flow Diagram....................................................36
4.4.1 Context Level........................................................39
4.4.2 Level–1 Data Flow Diagram.................................39
4.4.3 Level – 2 Data Flow Diagram..............................40
4.5 Data Dictionary..........................................................41

5 EXPERIMENTS AND RESULTS........................................45
5.1 Google Chrome Developer Tools.................................45
5.2 Network Panel..........................................................45
5.3 Experiment Setup......................................................46
5.4 Results Graph..........................................................48
5.4.1 3.0 Mbps Network.................................................48
5.4.2 6.0 Mbps Network................................................51
5.4.3 2.03 Mbps Network...............................................54
5.4.4 19.47 Mbps Network............................................57
5.4.5 100 Mbps Network...............................................60
5.4.6 Data Compression During Page Load of ClassTA .................................63

6 CONCLUSION............................................................................................................66

REFERENCES ........................................................................................................................68
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>Frequency Occurrence</td>
<td>22</td>
</tr>
<tr>
<td>2.2.</td>
<td>Huffman Code</td>
<td>25</td>
</tr>
<tr>
<td>2.3.</td>
<td>Huffman Code Representation of String</td>
<td>26</td>
</tr>
<tr>
<td>2.4.</td>
<td>Decoding a Huffman Encoded String</td>
<td>27</td>
</tr>
<tr>
<td>4.1.</td>
<td>Scripts</td>
<td>43</td>
</tr>
<tr>
<td>4.2.</td>
<td>Groups</td>
<td>44</td>
</tr>
<tr>
<td>4.3.</td>
<td>Huffman</td>
<td>44</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

PAGE

Figure 1.1. Web 2.0 map...........................................................................................................2
Figure 1.2. Web 1.0 system overview........................................................................................3
Figure 1.3. Web 2.0 system overview........................................................................................4
Figure 1.4. How AJAX works. ..................................................................................................7
Figure 1.5. Comparison between the classical and AJAX based web application model........8
Figure 1.6. ShrinkSafe online JavaScript compressor ............................................................16
Figure 2.1. Huffman tree.........................................................................................................25
Figure 3.1. ClassTA desktop...................................................................................................31
Figure 3.2. Desk for test-101. .................................................................................................31
Figure 4.1. Class diagram of Huffman Encoder. ....................................................................34
Figure 4.2. Class diagram of Decoder. ....................................................................................35
Figure 4.3. Use case diagram.................................................................................................37
Figure 4.4. Sequence diagram.................................................................................................38
Figure 4.5. Context (Level – 0 Data Flow ) diagram..............................................................39
Figure 4.6. Level-1 data flow diagram....................................................................................40
Figure 4.7. Level-2 Data flow diagram for encoding. ............................................................41
Figure 4.8. Level-2 Data flow diagram for decoding previously encoded JavaScript. .........42
Figure 5.1. Task manager........................................................................................................46
Figure 5.2. Web inspector.......................................................................................................47
Figure 5.3. Network panel of Chrome developer tools...........................................................47
Figure 5.4. 3.0 Mbps network for ClassTA’s Huffman version..............................................49
Figure 5.5. 3.0 Mbps network for ClassTA’s Shrink Safe version.........................................49
Figure 5.6. 3.0 Mbps network for ClassTA’s dynamic version............................................50
Figure 5.7. 3.0 Mbps network for ClassTA’s static version..................................................50
Figure 5.8. 3.0 Mbps network for ClassTA’s loadtime..........................................................51
Figure 5.9. 6.0 Mbps network for ClassTA’s Huffman version.............................................52
Figure 5.10. 6.0 Mbps network for ClassTA’s Shrink Safe version......................................52
Figure 5.11. 6.0 Mbps network for ClassTA’s dynamic version .............................................53
Figure 5.12. 6.0 Mbps network for ClassTA’s static version ..................................................53
Figure 5.13. 6.0 Mbps network for ClassTA’s loadtime .........................................................54
Figure 5.14. 2.03 Mbps network for ClassTA’s Huffman version .........................................55
Figure 5.15. 2.03 Mbps network for ClassTA’s Shrink Safe version ....................................55
Figure 5.16. 2.03 Mbps network for ClassTA’s dynamic version ..........................................56
Figure 5.17. 2.03 Mbps network for ClassTA’s static version ................................................56
Figure 5.18. 2.03 Mbps network for ClassTA’s load time .....................................................57
Figure 5.19. 19.47 Mbps network for ClassTA’s Huffman version .......................................57
Figure 5.20. 19.47 Mbps network for ClassTA’s Shrink Safe version ...................................58
Figure 5.21. 19.47 Mbps network for ClassTA’s dynamic version ........................................59
Figure 5.22. 19.47 Mbps network for ClassTA’s static version .............................................59
Figure 5.23. 19.47 Mbps network for ClassTA’s loadtime ....................................................60
Figure 5.24. 100 Mbps network for ClassTA’s Huffman version ..........................................60
Figure 5.25. 100 Mbps network for ClassTA’s Shrink Safe version ......................................61
Figure 5.26. 100 Mbps network for ClassTA’s dynamic version ..........................................62
Figure 5.27. 100 Mbps network for ClassTA’s static version ..............................................62
Figure 5.28. 100 Mbps network for ClassTA’s loadtime .....................................................63
Figure 5.29. Average loadtime of ClassTA on various networks ..........................................64
Figure 5.30. Data size during page load of ClassTA ..............................................................65
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Christopher Paolini, for all his guidance, direction, advice and encouragement throughout the time I was working on this thesis. He mentored and helped me develop good research ethics. I would also like to sincerely thank my committee members Dr. Subrata Bhattacharjee and Dr. Carl Eckberg for serving on my thesis committee and their comments towards improving this document.

Last but not the least; I would like to express my deepest gratitude for the constant support, understanding and love that I received from my family, friends and my girlfriend during past years.
CHAPTER 1

TECHNOLOGIES AND PREVIOUS RESEARCH WORK

This chapter is an introduction to the evolution of web. It briefly describes web technologies such as AJAX and JSON, which are currently the sought after technologies associated with web. It also gives details about various tools used for data compression.

1.1 Web 2.0

The primary function of a Web browser is to convert a request received by the server into network protocols and translate responses from the server in a user understandable format. Upon request, a server can supply any webpage using its intelligence. Web 2.0 encompasses the proliferation of interconnectivity of Web delivered content [1].

Figure 1.1 shows the Web 2.0 themes. The following are few examples of Web 2.0.

- Wikipedia - collaborative encyclopedia
- YouTube - upload video, comment, view videos
- LinkedIn - professional networking, resume 2.0, get references on your online resume
- Facebook – socializing with friends, various applications
- Yelp - local reviews
- Technorati/Digg/Reddit/Sphinn - social bookmarking; adding and voting for favorite news, blog posts, articles
- Upcoming/Eventful - share coming events
- Feedburner - sharing RSS feeds
- Pandora - find similar music to your favorites and checking who else likes it
- Flickr – sharing of images
- Twitter - messaging; old style IM doesn't count as web 2.0 because you can't see your friends' friends or what they're talking about with them- with twitter, you can- it's the web 2.0 version of IM [2]
Web 2.0 can be considered neither as a programming language nor a framework. Web 2.0 has several ways to make the Web page based mechanism smarter in performance and more elegant in appearance. The following are few Web 2.0 based components.

- Wiki
- CSS
- APIs on XML or JSON framework
- AJAX

Figure 1.2 displays Web 1.0 technology where the file server is the fundamental unit of the system. Here the Internet is used for the communication between the consumer and the server. The unidirectional arrows going from server to hosts depicts that the server to host is a one-way communication. The consumer cannot contribute or give inputs to the server. Exceptionally, if one wishes to contribute, the contributor has to submit its text to a web site administrator and the administrator has to put it on the server. So this central server system
Web 1.0

HTML pages

few Content creators

media files

software distribution

file server

Internet

many Content consumers
discussion group

hosting service

e-mail

HTML

slow connections

stores & snail mail

Figure 1.2. Web 1.0 system overview.

does not remain real time. If this architecture was supporting server side changes from the client, changes could have been reflected much faster with respect to time.

Figure 1.3 depicts the system structure for a Web 2.0 system. The major difference between both is the unidirectional arrows and bidirectional arrows shown in the respective figures. Every individual host machine in a Web 2.0 framework acts as a client as well as a contributor unlike in Web 1.0. Taking an example of a wiki, we know that the user can make changes directly without involving the administrator of the system.
1.2 JavaScript

There is a wide use of JavaScript in Web-based applications and it is increasingly popular with developers. JavaScript is a widely used programming language that enables new generation of computer applications. Scripting languages are used to develop a large amount of Web content. Many websites such as Google and Yahoo rely heavily on JavaScript to create pages that are more dynamic, attractive and responsive. Vendors of every major browser including Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer are apprehensive about the performance of JavaScript as it is so widely used to enable Web 2.0
applications. JavaScript implementations based on Just-InTime (JIT) compilation strategies are a result of the strong competition between major vendors, also known as the browser wars. The browser market share is highly important to companies competing in the Web Services market, so an objective comparison of the performance of different browsers is valuable to both consumers and service providers [3].

In contrast to what their names suggest, Java and JavaScript are entirely different. Java is a class based object-oriented language whereas JavaScript is a prototype based language, which is heavily influenced in design by itself. JavaScript has become highly popular because of its standardization, availability in every browser implementation and tight coupling with the browser’s Document Object Model. JavaScript is a light weight language which is ideally suited for data validation, interactive Web pages, and improved Web navigation. Fortunately, JavaScript offers high opportunities for optimization of file size and execution speed using techniques like packing, compression, and obfuscation. Due to these features, there could be savings of around 50 to 90 percent off the size of JavaScript files. The following are ways to achieve the aforementioned.

- Trim the downloadable file size by removing extra whitespace and comments and using semicolons to avoid misunderstandings.
- Reduce HTTP requests by combining files and merging or embedding scripts on high traffic pages.
- Defer or delay loading where possible.
- Load only what's necessary, segment API code (NS4, IE5, DOM, and so on).
- Compression of the larger external files located in the head section of a document.
- Abbreviate and map names automatically or manually to make your optimization reversible.
- Crunch and obfuscate your code to shrink scripts and deter reverse engineering.
- Create self-extracting archives if entering the 5K competitions.
- Use ESC or Blue Clam to abbreviate variable and object names for maximum packing.

Blue Clam is A Java-based JavaScript obfuscator from Solmar Solutions. Supports recursive directory tree parsing, a user-defined keyword dictionary, variable length obfuscated keyword support, extended file types (such as .js, .jsp, and .asp) and a graphical environment [4].
ESC (ECMA Script Cruncher) is an ECMAScript pre-processor written in JScript. In addition to removing whitespace and comments from JavaScript, it can optionally rename variables in JavaScript.

The 5K page contestants employ self-extracting archives to save space at the expense of time.

1.3 JSON FRAMEWORK

JSON stands for JavaScript Object Notation. It was developed in 1999 as a substitute to XML. JSON is a data interchange format. It is text based and is used to represent JavaScript objects as strings and transmit them over a network. It is generally used in AJAX applications. JSON text is easy to produce and read. Parsing JSON is generally faster compared to XML. Hierarchical data could be easily represented using it. Being lightweight makes it an ideal choice for AJAX applications [5].

1.4 AJAX

AJAX stands for Asynchronous JavaScript and XML [6]. AJAX is not a new programming language, but a new way to use existing standards. Using AJAX, one can fetch data from the server and display the data in the browser without reloading the page. It only updates the section where the changes are required and not the whole page. Traditionally, Web pages (which did not use AJAX) reloaded the entire page if the contents needed to be changed. AJAX allows Web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes. Examples of applications using AJAX are Google Maps, Gmail, YouTube, and Facebook [7, 8].

Figure 1.4 shows how AJAX works. AJAX uses a JavaScript object called XMLHttpRequest for server communication. Almost all of the browsers support this object and allow connection to the server using this object. The communication can either be synchronous or asynchronous. In case of asynchronous calls, a client makes calls to the server and attaches callback methods and objects along with the call. After the initiation of the call, either client code precedes its sequential interpretation or just pauses and waits for callback method to be executed. The callback method should have remaining commands or actions to reflect the response received from server. In case of a synchronous call, execution
of scripts stops until a response from the server is received. If the time taken by the server to respond to the request is higher than some predefined threshold value, then a client code can time-out the request and continue with its execution. One of the key benefits of using AJAX is that it doesn’t place any restriction on the client code. Though JavaScript is highly recommended as a front end for AJAX based applications, Adobe Flash and Flex are also options used to develop AJAX based interfaces.

Figure 1.5 shows activity between a client and server in a classical Web application and an AJAX based Web application. In the later one, on client side, JavaScript gives complete flexibility to design user-friendly interfaces and most of the operations and control flow are handled on the client side. In an ideal scenario, anything related to business logic stays on the server. A high amount of flexibility is shown starting from high level programming languages like Java, C# or VB to create Web Services, to conventional server side programming with JSP, ASP, and PHP. Despite the long name of the term AJAX (Asynchronous JavaScript and XML), it minimizes down to a simple meaning; displaying the content of a Web page without having to reload it [9].

The principle steps that involve the usage of AJAX are:

- Capture an event (such as when a user changes a combo box entry or presses a button).
The event triggers JavaScript code, which sends a query to the Web server.

The JavaScript code retrieves results from the server.

The JavaScript code uses the results to modify the Web page’s content.

JavaScript accesses the Document Object Model (DOM) to modify the contents of a Web page without reloading it. The core elements of an AJAX page are the primary functions named getHTTPObject and handleHttpResponse. They are responsible for handling the page requests. The page request provides an input to the handleHttpResponse function and thus it extracts the information required in order to change an element in the page. It becomes quite a difficult task to write an AJAX code from scratch, especially when dealing with all different types of browsers such as FireFox, IE, Google Chrome. Luckily a surplus number of AJAX libraries exist that can manage the differences between the browsers [10]. With the
right design in place, there are many secondary strategies for improving performance such as minification of the code, HTTP compression, using CSS sprites, etc. [3, 11]

Minification is the process of removing all unnecessary characters from source code, without changing its functionality. These unnecessary characters usually include white space characters, new line characters, comments, and sometimes block delimiters, which are used to add readability to the code but are not required for it to execute.

CSS sprites is a CSS technique that can make your website load faster. Instead of having a dozen images that are included on every page of the site (one for your logo, one for an RSS button, a few fore social media links, etc.) you combine them all into one big image. You can then use CSS to display just the portion you need in any given spot.

1.5 Technique for High Performance Data Compression

All the data that is stored or transferred over communications links generally contains a significant amount of redundancy. There exist a few problems with compression techniques. For example,

- Poor runtime execution speeds interfere in the attainment of very high data rates
- Most compression techniques are not flexible enough to process different types of redundancy
- Blocks of compressed data that have unpredictable lengths present storage space management problems.

So the type of redundancy along with tolerance to the above-mentioned problems forms a basis for choosing between various compression-techniques [12, 13].

1.5.1 Types of Redundancy

There exist four types of redundancy in commercial data. Redundancy here is restricted to what is found in a data stream (without concerning how the data is to be interpreted). For illustration, types of redundancy will be described as they might be found in two kinds of files: English text and inventory records i.e. database. The redundancy categories described below are not independent, but overlap to some extent. The intent in showing the types of redundancy is not precisely the analysis of its components of it, but
rather to provide a better view of the opportunities and challenges for a compression algorithm.

1.5.1.1 CHARACTER DISTRIBUTION

In a particular string, some characters are used more often than others. Specifically, in eight-bit ASCII representations nearly three-fourths of the possible 256-bit combinations may remain unused in a specific file. In English text, the characters occur in a well-documented distribution, with 'e' and "space" most frequently used [14].

1.5.1.2 CHARACTER REPETITION

When a string consists of redundant characters, the message could be encoded in a more compact form. Such strings are not so common in text, occurring as blank spaces in place of tabs or at the end of lines. However, in business files, unused spaces are very common. An inventory record will frequently have strings of blanks in partially used alphabetic fields, strings of zeros in high-order positions of numeric fields, and perhaps strings of null symbols in unused fields.

1.5.1.3 HIGH USAGE PATTERNS

Certain sequences or patterns of characters in a string will appear with relatively high frequency and can therefore be represented using relatively fewer bits. In English text, a period followed by two spaces is more common than most other three-character combinations and could therefore be recoded to use fewer bits. Similarly, unlikely pairs, such as GC, would be encoded with very long bit combinations to achieve appropriate utilization of bits.

1.5.1.4 POSITIONAL REDUNDANCY

If certain characters appear consistently at a predictable position in each sample of data, then they are said to be partially redundant. Text has potentially none or less positional redundancy. An instance of raster-scan can be considered as a good example of positional redundancy. Suppose a raster-scan contains a vertical line and appears at the same position in each scan and thus could be more compactly coded. In inventory files, certain record fields
may almost always have the same entry, such as a "special handling" field which almost always has "none" in it.

1.5.2 Various Methods of Data Compression

The previous discussion of redundancy types provides a basis for comparing various compression methods [14].

1.5.2.1 Huffman Encoding

The standard Huffman encoding procedure describes a way to assign codes to input symbols. It works in a way such that each code length in bits is approximately $\log_2 (\text{symbol probability})$ where symbol probability is the frequency of occurrence of a particular symbol (The frequency is expressed using probability). For example, if the set of symbols is chosen to be the one-byte ASCII characters and if a character ‘a’ appears one-eighth of the time, then it is encoded into a three-bit symbol. If the letter Z is found to occur only very small percent of the time, 10 bits represent it. In normal use, the size of input symbols is limited by the size of the translation table needed for compression. Single-character encoding can cope with character distribution redundancy, but not the other types. Compression could be improved by using input symbols of two bytes each, but that would require a table of 64K entries at a cost that might not be warranted.

1.5.2.2 Run Length Encoding

Sequences of same or identical characters can be encoded using a count field with an identifier of the repeated character. This approach is very effective in graphical images, has almost no value in text, and has average value in data files. The problem pertaining to this approach is distinguishing the count field with normal characters which has the same bit pattern. The problem with run-length encoding for character sequences intermixed with other data is in distinguishing the count fields from normal characters, which may have the same bit pattern. This problem has several solutions, but has its own disadvantages.

1.5.2.3 Programmed Compression

The programming is usually performed by the applications programmer or data management system. In formatted data files, several techniques are used. Unused blank or
zero spaces are eliminated by making fields variable in length and using an index structure with pointers to each field position. The field values which are predictable are compactly encoded by way of a code table. Each field has its own specialized code table that deals with positional redundancy. Since programmed compression cannot effectively handle character distribution redundancy, it is a nice complement to Huffman coding.

1.5.2.4 ADAPTIVE COMPRESSION

Here, variable-length strings of input symbols are converted into fixed-length codes. The symbol strings are selected so that all have almost equal probability of occurrence. Eventually, strings of frequently occurring symbols will have more symbols than a string having infrequent symbols. This kind of compression is effective in dealing with character frequency redundancy, character repetitions, and high-usage pattern redundancy. However, it is not that effective on positional redundancy. This type of algorithm is called to be adaptive because it initially has an empty table of symbol strings and builds the table during the processes of compression and decompression [15].

1.6 BASE64 ENCODING

There are some environments that are restricted to using US-ASCII data. This is an area where Base Encoding is used. It is also used in new applications pertaining to its property to manipulate objects with text editors. In the past, applications have had different requirements and thus sometimes implemented base encodings in different ways. In today's scenario, protocol specifications sometimes make use of base encodings in general, and "base64" in particular, without a precise description or reference. The Base 64 encoding is designed in a way to represent arbitrary sequences of octets in a form that requires case sensitivity but does not need to be in a human readable format. It uses 65-character subset of US-ASCII, wherein 6 bits are used to represent a single printable character. The extra 65th character, "=" is used to specify a special processing function. The encoding process presents 24-bit groups of input bits as output strings of 4 encoded characters. Starting from left to right, a 24-bit input group is formed by concatenating three 8-bit input groups. These 24 bits are then considered as 4 concatenated 6-bit groups, each being converted to a single digit in
the base 64 alphabets. Each 6-bit group is used as an index into an array of 64 printable characters. The character referred by the index is placed in the output string [16].

Base64 is a group of encoding schemes that displays binary data in an ASCII format by converting it into a radix-64 representation. The Base64 term evolves from a specific MIME content transfer encoding. This scheme is basically used where there is a requirement to encode binary data that needs be stored and transferred over communication media that is originally designed to deal with textual format of data. It ensures consistency of data during transport. Base64 is used commonly in a number of applications such as email via MIME and storing complex data in XML [17].

1.7 WEB WORKERS

Web Workers are the latest feature evolved in current versions of Web browsers. Web Workers allow JavaScript to run parallel on a web page, without blocking the user interface. In order to achieve any type of computation using JavaScript one needs to break tasks into smaller parts and split their execution apart using timers [18]. This type of execution is extremely slow. The Web Worker recommendation is partially based on the prior work done by the Gears team on their WorkerPool Module. The idea has since grown and has become a full recommendation. A 'worker' is a script that will be loaded and executed in the background. Web Workers provide a way to do this task without much effort. One can communicate with the worker using messages. All browsers support passing of a string message (Firefox 3.5 also supports passing in JSON-compatible objects). This message will be communicated to the worker and the worker can also communicate back with the parent page. Workers allow considering a general piece of computation and parallelizing it to a great extent. In this way, there would exist two versions of a script (one that runs in older browsers and one that runs in a worker) [19].

1.8 JAVASCRIPT COMPRESSION TOOLS

In terms of code minification, the most known tools to minify JavaScript code are Douglas Crockford's JSMIN, the Dojo compressor and Dean Edwards' Packer. Each of these tools, however, has its pros and cons.
1.8.1 YUI Compressor

The objective of JavaScript and CSS minification is to preserve the operational features of the code while reducing its overall size. The YUI (Yahoo User Interface) Compressor is open-source [20]. It is designed to be 100% safe and yield a higher compression ratio than most other tools. Various tests on the YUI Library have proven that the amount of savings of it is over 20% as compared to JSMin. Starting with version 2.0, the YUI Compressor is also able to compress CSS files by using a port of Isaac Schlueter's regular-expression-based CSS minifier. The YUI Compressor is written in Java. It relies on Rhino to tokenize the source file. Rhino is an open-source implementation of JavaScript written entirely in Java. It is typically embedded into Java applications to provide scripting to end-users [21]. At first, it starts to analyze the source JavaScript file to understand its structure. It then prints the token stream, omitting as many white spaces as possible, and replacing all local symbols by a 1 to 3 letter symbol wherever such a substitution is appropriate. A set of finely tuned regular expressions is used by the CSS compression algorithm to compress the source CSS file [20].

1.8.2 JSMin

Douglas Crockford writes JavaScript minifier. It is a filter which removes comments and unnecessary whitespace from JavaScript files. It typically reduces file size by half, which eventually results in faster downloads. It omits or changes some characters. This doesn’t affect the behavior of the program that it is minifying. The result may be harder to debug and would definitely be harder to read. JSMin first changes carriage returns (\r') to linefeeds (\n'). It replaces all other control characters (including tabs) with spaces. It replaces comments in the // form with linefeeds. It replaces comments in the /* */ form with spaces. All runs of spaces are replaced with a single space. All runs of linefeeds are replaced with a single linefeed. JSMin does not change quoted strings and regular expression literals. JSMin does not complicate, but it does make the code difficult to read. It expects the character set encoding of the input program to be either ASCII or UTF-8. It does not work perfectly with other encodings [3, 22].
1.8.3 Packer

One of the popular tool for **compressing, minifying and packing your code** on demand is Dean Edwards packer tool [23]. It gives a choice whether to **compress** (i.e strip excess space and comments), **minify** (i.e shorten) or **pack** (i.e encode the function so that it has to be run through an eval statement for it to run) a given set of code. Packer reduces the size of the original Javascript code as well as makes it difficult for anyone to interpret the original code.

1.8.4 Shrink Safe

Dojo is a JavaScript framework, a collection of utilities written to ease development of client-side applications [24]. Dojo provides a compressor that can be used on one’s own code. ShrinkSafe is a JavaScript "compression" system which is used by Dojo [25]. It can typically reduce the size of scripts by one third or more, depending on programming style. ShrinkSafe is based on Rhino, a JavaScript interpreter which allows transforming the source of a file in a way such that the resulting script will function identically to the file that has been uploaded. The best part of Shrink Safe is that it has to never change a public variable or API. It means that you can drop the compressed version of your JavaScript into your pages without changing the code that uses it. ShrinkSafe can make pages respond faster by reducing the number of HTTP requests needed and by decreasing the size of the files served. In tests on the Swat JavaScript code base, ShrinkSafe produced files that were 36 percent smaller after gzip compression. ShrinkSafe (and the Rhino engine) are written in Java. The Dojo Toolkit distributes ShrinkSafe as a Jar file. One can also build ShrinkSafe using Mozilla CVS and the patch provided by the Dojo Toolkit. It also needs to have a Java SDK installed as well as the Apache Ant build system [25].

Figure 1.6 shows the ShrinkSafe online JavaScript compressor. This tool is used for compressing ClassTA’s JavaScript files. After compression, compressed files are stored in the database. When client requests for ClassTA’s shrink safe version, all compressed files get called and loaded. The aim is to compare load time of JavaScript files against ClassTA’s Huffman approach.
1.8.5 Comparison

JSMin was RegEX based, we would periodically hit issues where it would choke abort on a certain complex JS expression. It would break the working code in ways that only appeared at run-time which meant that it would be difficult to debug or reproduce issues until the users found them. After evaluating its features, it could be considered as a good fit. It is based on Rhino so was a true JS parser and didn’t alter working code. In addition to removing whitespace and comments like JSMin, it would rename and reduce the length of internal variable and function names. So it really compressed the javascript code in a way that eventually helped reduce load time. Julien Lecomte has written the YUI Compressor inJava that uses Rhino to parse the JavaScript. It reaches a higher level of compression than the Dojo compressor. One of the primary reasons is that it will reuse the short tokens instead of increasing counters that are used by ShrinkSafe. The YUI Compressor also compresses CSS and so it can be used to help reduce the size of our CSS files that we deliver to the browser [26].
CHAPTER 2

HUFFMAN ENCODING

There has been a lot of research in the field of data compression. Many techniques and standards have been developed in this field to compress data efficiently and effectively. Data compression can be defined as reducing the amount of storage space required to store a given amount of data. The primary goal behind data compression is to represent the given data as accurately as possible using fewer numbers of bits [27].

Compression consists of encoding data and information in fewer bits than the normal data (not encoded) would take. There are many advantages pertaining to this technique. First of all, using fewer bits for data would save the storage space and so you can efficiently use it for some other purpose. Secondly, the transfer of files becomes faster which leads to a faster response to the user’s request and the bandwidth requirement is also less. But along with the benefits, there are some shortcomings of this technique such as the data once compressed, needs to be decompressed while needed and this decompression requires some extra processing as well as resources.

A simple example of data compression involves transforming a steam of characters in some representation (ASCII) into a new string (bits), which is basically smaller in length but contains the same data. Data compression plays an important role in applications which deal with data transmission and data storage. Data processing applications have always been in high demand. These are the applications that require large volumes of data and hence similarly larger storage space. Moreover, distributed systems are high in demand. These applications require a huge amount of data to be transferred over communication links. Compressing data before storage or transmission reduces storage as well as communication costs. When the amount of data to be transmitted is reduced, capacity of the communication channel could be increased. Similarly, compressing a file is equivalent to increasing the capacity of the storage medium. It may then become viable to store data at a higher, and thus a faster, level of the storage hierarchy and reduce the load on the input/output channels of the computer system.
2.1 DATA COMPRESSION ENCODING TYPE

Data Compression is broadly classified into two main categories namely Lossless and Lossy Compression techniques. 2.1 and 2.2 gives a detailed explanation of both the categories.

2.1.1 Lossless Compression Techniques

Lossless data compression is a type of data compression in which the recovered data is assured to be identical to the source. The data before the compression and after the decompression are alike. The categorizations of these compression methods are done on the basis of the data they are designed to compress. The types of data generally are text, executable files, images, and sound. Usually, a general-purpose lossless compression algorithm works on any type of data. But sometimes it does not accomplish significant compression with the data except the one it was originally designed for. For example, sound data cannot be compressed well with a traditional algorithm which was originally developed for text data. Lossless compression algorithms usually make use of the statistical redundancy in such a way so that it can present the sender’s data concisely and perfectly. Due to the existence of statistically redundant data in real-world, lossless compression becomes possible. Considering an example in English language, the letter ‘e’ is much more common than letter ‘z’, and the probability that the letter ‘z’ will be followed by the letter ‘q’ is very small. Most of the lossless compression programs use two varied algorithms. The first one generates a statistical model for the data to be inputted, and second one maps the input data to bit strings using a model in a way that more frequently used data will produce shorter output than less frequently used data [28, 29]. Statistical modeling algorithms for text (or text-like binary data such as executables) include:

- Burrows-Wheeler transform
- LZ77

LZW (Lempel–Ziv–Welch) encoding algorithms to produce bit sequences are:

- Huffman coding (also used by the Deflate algorithm)
- Arithmetic coding

Michael Burrows and David Wheeler released a research report in 1994 discussing work they had been doing at the Digital Systems Research Center in Palo Alto, California.
Their paper, "A Block-sorting Lossless Data Compression Algorithm" presented a data compression algorithm based on a previously unpublished transformation discovered by Wheeler in 1983. The BWT is an algorithm that takes a block of data and rearranges it using a sorting algorithm. The resulting output block contains exactly the same data elements that it started with, differing only in their ordering. The transformation is reversible, meaning the original ordering of the data elements can be restored with no loss of fidelity [30].

Jacob Ziv and Abraham Lempel had introduced a simple and efficient compression method. This algorithm is referred to as LZ77 in honour to the authors and the publishing date 1977. LZ77 is a dictionary based algorithm that addresses byte sequences from former contents instead of the original data. In general only one coding scheme exists, all data will be coded in the same form:

- Address to already coded contents
- Sequence length
- First deviating symbol

If no identical byte sequence is available from former contents, the address 0, the sequence length 0 and the new symbol will be coded [31].

LZW and its variants are generally implemented in open-source and proprietary software. Usage of some algorithms requires authorization as these are patented by individuals or groups in the USA and other countries. Due to the patents on certain kinds of LZW compression algorithms, some open source activists have encouraged and inspired people to avoid using the Graphics Interchange Format (GIF) for compressing image files in favor of the Portable Network Graphics PNG format, which combines the LZ77-based Deflate algorithm with a selection of domain-specific prediction filters [32]. Many of these techniques are used for text data which also work well for image data. There are also such techniques which do not work for common text but are quite advantageous with images. Specifically, bitmaps and other techniques make use of special characteristics of images, such as the common phenomenon of contiguous 2-D areas of similar tones, and the fact that color images generally have prevalence to a restricted range of colors out of those that are representable in the color space. As shown in the case of indexed images, sound data could also have some prevalent characteristics, which could be used while designing algorithms for them Lossless sound compression algorithms can make use of the repeating patterns shown
by the wave-like nature of the data, which is essentially used by models to predict the next value, and encoding the difference between the expected value and the actual data. The difference between the predicted and the actual data, which is named ‘error’ generally, comes out to be of a small value. Due to this fact certain difference values like 0, +1, -1 etc. on sample become very frequent, so such values could be encoded using the least number of bits. This is called delta coding.

### 2.1.2 Lossy Data Compression

In a lossy data compression method, the data which is compressed and then decompressed is not same as the original data and has a slight difference which is tolerable. This type of technique is generally used in streaming media and telephony applications. In this context, these methods are typically referred to as codecs. Most of the lossy data compression methods suffer from generation loss. Generation loss refers to a condition wherein repeatedly compressing and decompressing a file will lead to a progressive loss of data. This is in contrast with lossless data compression [33]. There are two basic lossy compression schemes:

1. In lossy transform codecs, pictures or sound samples are chopped into small segments, transformed into a new basis space, and quantized.
2. In lossy predictive codecs, prediction of a current sound sample or image frame is done using previous and subsequent decoded data. The error between both of them together, with any extra information needed, is used to reproduce the prediction. Data is then quantized and coded.

### 2.1.3 Lossless vs. Lossy Compression

Lossy files are advantageous compared to lossless methods in a way such that it produces a much smaller compressed file than any known lossless method while still meeting the requirements of the application. Lossy methods are generally used with sound, images, or videos. The compression ratio, (i.e the size of the compressed file compared to that of the uncompressed file) of lossy video codecs is always greater than those of the audio and still-image equivalents. The ratio at which audio can be compressed with a bare minimum loss of quality is 1:10 (where 10 is the size of uncompressed file and 1 is the size of compressed file) where as video could be compressed fully without any noticeable change at a rate of 1:300.
Lossy compression methods compress a still image to 1/10\textsuperscript{th} their original size, same as with audio, but the quality loss is more noticeable, specifically on close inspection. When the user receives a file that is compressed by a lossy compression method, it is quite different from the original file at the bit level but does not makes a difference as it is quite undistinguishable to human eye or ear. Many methods are focused on the idiosyncrasies of the human anatomy, considering certain features as that the human eye can see only certain range of the frequencies of light. Flaws caused in lossy compression, which are visible to the human eye or ear, are known as compression artifacts. For example, a person viewing a picture or television video scene might not notice if some of its finest details are removed or not represented perfectly. Similarly, two clips of audio may be perceived as the same to a listener even though one is missing details found in the other. Lossy data compression algorithms introduce relatively minor differences and represent the picture, video, or audio using fewer bits. While these methods are reverted, data could be achieved back completely or it may not be achieved. While we reverse lossless compression data, original data can be reconstructed; which may not be possible in case of lossy schemes, it may not be achieved. Attempts to compress data that has been compressed already will therefore usually result in an expansion, as will attempts to compress encrypted data.

2.2 FIXED LENGTH VS. VARIABLE LENGTH CODING

Data encoding schemes broadly fall into two categories, fixed length encoding and variable length encoding. Fixed length encoding deals with encoding all the characters with the same number of bits and variable length encoding deals with encoding based on priority. Suppose we have 8 symbols to be encoded in a fixed length encoding, an approach will be to use three bits to encode 8 symbols ($2^3 = 8$). The best example of fixed length encoding is ASCII code, which uses 7 bits to encode a total of 128 different symbols, the 8\textsuperscript{th} bit is used as a parity bit. One of the drawbacks of fixed length codes is that the code never considers the probability of occurrence of the symbols to be encoded. A character that appears 500 times will be encoded with the same number of bits as a symbol which occurs only 10 times. This makes fixed length encoding inefficient for data compression. Variable length encoding overcomes this drawback by using a fewer number of bits for characters which appear more frequently, and more bits for symbols which occur less frequently [34].
2.3 Huffman Algorithm

D.A Huffman developed the Huffman Encoding algorithm in 1950 [35]. Huffman coding is a statistical technique which attempts to minimize the number of bits used for representing a string of characters or symbols. The goal of this algorithm is fulfilled by allowing the encoded symbols to vary in length. Longer codes are applied to those symbols whose occurrence is less in the string and similarly shorter codes are designated to those whose occurrence is more [36]. Huffman coding is classified into two different categories: Static Huffman coding and Adaptive Huffman coding. Static Huffman coding uses statistical symbol frequency tables, wherein the symbol frequencies are known in advance (i.e. we know which symbol will appear how many times before we start the actual encoding process). In contrast, while using Adaptive Huffman compression, we need not know the frequency in advance. The symbols are encoded as encountered [37]. Huffman coding can be considered as an efficient way for compressing a file. Before actually starting the compression of a file, we may actually scan through the complete file and determine the number of occurrences of a particular symbol. After we have generated a symbol frequency table we can apply the encoding algorithm and get the compressed codes for the symbols. Table 2.1 shows an example of a symbol frequency table of a text file that contains 100 characters in total. Below is a sample text string used as an example.

“abcdefabcdefabcdefabcdefabcdeabcdeddddccccbbbaaabbbbbbbaabbbbbbbbaaaaaaaaaaanaaaaaaaaaaaaaaa”.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency of occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>40</td>
</tr>
<tr>
<td>b</td>
<td>30</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
</tr>
<tr>
<td>e</td>
<td>6</td>
</tr>
<tr>
<td>f</td>
<td>4</td>
</tr>
</tbody>
</table>
Often, a universal statistical symbol frequency table is prepared which is derived from the earlier experiences and analysis of various input streams, (e.g. in most of the words used in the English language the letter ‘a’ appears more frequently than the letter ‘z’). Once a symbol frequency table has been constructed, we can proceed with the algorithm to compress the data. To get the encoding for a set of symbols, we first need to make a Huffman tree out of the symbols. A Huffman tree is an example of a strict binary tree in which all nodes except the leaf nodes have both right and left children. The leaf nodes of the tree are the actual symbols to be encoded. One of the characteristics of a strictly binary tree is that if there are \( n \) leaf nodes (symbols to be encoded) then there will be a total of \( 2^n - 1 \) nodes in the tree. The following algorithm will create the Huffman tree from the given set of symbols and will assign codes to the tree created:

Algorithm for creating a Huffman tree:

1. Input all symbols (characters) along with their respective frequencies.
2. Create leaf (right and left links will be NULL) nodes representing the symbols scanned.
3. Let \( S \) be the set containing all the nodes created in step 2.
4. [Create the Tree]
   While there is only one node in \( S \)
   1. Sort the nodes (symbols) in \( S \) with respect to their frequencies.
   2. Create a new node to combine the 2 nodes (symbols) with least frequencies.
   3. Frequency of this new combined node (symbol) will be equal to sum of frequencies of nodes (symbols), which were combined. This newly created combined node (symbol) will be parent of 2 nodes, which were combined.
   4. Replace, in \( S \), the 2 nodes (symboles), which were combined with the new combined node (symbol).

[After the 4\(^{th}\) step one will be left with only one node, which in turn is the root of the Huffman tree, having frequency equal to the sum of all frequencies of all symbols and a tree is generated with leaf nodes containing the basic symbols whose code is to be found.]

[Assigning the codes to the tree]

Start from root of tree generated in step 4 and go on assigning 1 to every left branch and 0 to every right branch. Let us take an example that will make things clearer; we will create a Huffman tree for the symbols in Table 2.1 (p. 22) given earlier.
Figure 2.1 shows Huffman tree. First we create the leaf nodes from the symbol frequency table and sort them. Symbols ‘f’ and ‘e’ have the least frequencies; 4 and 6 respectively; these 2 nodes are combined to make a node ‘fe’ having frequency 4+6=10. This new node ‘fe’ is now the parent node of the nodes ‘f’ and ‘e’, and ‘fe’ replaces ‘f’ and ‘e’. Again we sort the nodes; now ‘fe’ and ‘c’ have least frequencies i.e. 10 each. This time we combine ‘fe’ and ‘c’ to create a new node ‘fec’ having frequency 20. Nodes ‘fe’ and ‘c’ are replaced be their parent ‘fec’. After sorting again; we now combine ‘d’ and ‘fec’ to create ‘dfec’ having frequency 30, 10 of ‘d’ and 20 of ‘fec’. This ‘dfec’ becomes parent of ‘d’ and ‘fec’ and replaces both of them. It is now time to combine the nodes ‘dfec’ and ‘b’ as they have least frequency. The new node will be ‘dfecb’ having frequency of 60; again ‘dfec’ and ‘b’ will be replaced by ‘dfecb’. Now only two nodes are left namely ‘dfecb’ and ‘a’. We again sort $S$ and combine both of the remaining nodes to form ‘adfecb’ which has frequency count of 100. After making ‘adfecb’ parent of ‘a’ and ‘dfecb’ and replacing them with ‘adfecb’; we have created the Huffman tree for the symbols in Table 2.1 (p. 22). Node ‘adfecb’ is the root of the tree. Assigning codes to the tree created is still to be done. To assign the codes start from the root of tree and go on assigning 1 to every left branch and 0 to every right branch, after which the tree creation part is complete. Now the question that arises is how to find the code for a particular symbol? We can find the code for the specific symbol starting from the bottom most symbol and traversing up in direction towards the root. As soon as you encounter an individual branch output, the code assigned to that branch (either 0 or 1) could be noted down; when you reach the root you will have the code for all the symbols from LSB to MSB. Suppose we need to find Huffman code for ‘f’; we start from ‘f’ and move towards the root i.e. f -> fe -> fec -> dfec -> dfecb -> adfecb; we get code 1, 1, 0, 1, 0 respectively (i.e. 1 for f->fe , 1 for fe -> fec, 0 for fec -> dfec and so on). We should consider that this code which we have is from LSB to MSB so the final code for ‘f’ will be “01011”. Table 2.2 shows Huffman code for all the symbols.

After the table is constructed, we can conclude no code for one symbol matches another and also it does not work as a prefix for another symbol’s codeword. (e.g. the code for 'b' is 00; and there is no other code that begins with 00). Codes with this property are called prefix codes. In prefix codes no code in the set is a prefix to the code for another symbol. This property makes Huffman decoding easier when it is required. For an example,
Figure 2.1. Huffman tree.

Table 2.2. Huffman Code

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency of occurrence</th>
<th>Huffman Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>40</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>C</td>
<td>10</td>
<td>0100</td>
</tr>
<tr>
<td>D</td>
<td>10</td>
<td>011</td>
</tr>
<tr>
<td>E</td>
<td>6</td>
<td>01010</td>
</tr>
<tr>
<td>F</td>
<td>4</td>
<td>01011</td>
</tr>
</tbody>
</table>
we want to transmit ‘abdceabedf’; we can directly send Huffman code for the every symbol from Table 2.2 (p. 25) (i.e. 1 for ‘a’, 011 for ‘d’ and so on). Table 2.3 shows what will be transmitted for each of the symbols.

Table 2.3. Huffman Code Representation of String

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>d</th>
<th>c</th>
<th>e</th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00</td>
<td>011</td>
<td>0100</td>
<td>01010</td>
<td>1</td>
<td>00</td>
<td>01010</td>
<td>011</td>
<td>01011</td>
</tr>
</tbody>
</table>

So the code being transmitted will be ‘100011010001010010010111101011’. The total number of characters in the string is 10 and 31 bits are used to transmit them. If we use the normal ASCII code instead (i.e. 7 bit fixed length code) then it will take 7*10=70 bits to transmit the same string. So in this case, using Huffman code saved 39 bits, which is around 55%. It comes as a conclusion that Huffman codes can save from around 20% to 90% depending pattern used for compression.

2.4 DECODING HUFFMAN CODES

Decoding deals with the reverse process of identifying the symbols from the code. As mentioned earlier, Huffman compression is a lossless compression scheme, so it is possible to retrieve symbols. We need the Symbol Frequency Table, which was used at the time of compressing the data for decoding the Huffman codes. We cannot recover the data back without using the Symbol frequency table. The decoding the Huffman code starts with the same step as that we did in the earlier step of encoding. The first step involved in decoding Huffman codes is same as the one used in finding the Huffman codes (i.e creation of Huffman Tree from the symbol frequency table). We get the same tree that we developed while compressing the data. For decoding Huffman codes, we make use of the above mentioned property of the Huffman codes (i.e. Huffman codes are prefix codes). Once we have the Huffman tree as in the above figure, we can easily reverse the data and receive it back.

The algorithm used for decoding the Huffman codes is:

- Start from the root of the tree
- Examine the next element in the input
• If it is 1, move to the left child.
• If it is 0, move to the right child.
• If it is a leaf node
  • Output its symbol
  • Go to step 1
• If it is not a leaf node then go to step 2

Suppose we want to decode the stream ‘1001101000101000101001101011’ for the above created Huffman tree. We start from the root (node ‘adfecb’), as first bit is 1 we go to the left child which is node ‘a’; now this ‘a’ is a leaf node so we output the symbol ‘a’ and again start our way from the root. This time input is 0, so we move to right child of the root (i.e. node ‘dfecb’); this node is not a leaf node so we continue our search. Again next input bit is 0 so we again visit the left child of node ‘dfecb’ which is node ‘b’; now node ‘b’ is a leaf node so we output the symbol ‘b’ and restart the search. Table 2.4 shows symbols for Huffman string.

Table 2.4. Decoding a Huffman Encoded String

<table>
<thead>
<tr>
<th>1</th>
<th>00</th>
<th>011</th>
<th>0100</th>
<th>01010</th>
<th>1</th>
<th>00</th>
<th>01010</th>
<th>011</th>
<th>01011</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>d</td>
<td>c</td>
<td>e</td>
<td>A</td>
<td>b</td>
<td>e</td>
<td>d</td>
<td>f</td>
</tr>
</tbody>
</table>

So the final output symbol string is ‘abdceabedf’.

Implementation of Static Huffman Encoding faces a few setbacks and some of them are mentioned:

• If we consider a network in which Huffman compression is applied on the data, it generally deals with the functioning at some lower layer (e.g. physical layer) where Huffman compression is applied before actually sending data on the communication channel. A constraint according to the Huffman’s algorithm is that the symbol frequencies must be known in advance. This could be a real problem considering that we cannot build the Symbol Frequency table in advance as in most of cases we do not know the nature and type of data being transmitted (i.e. we do not know whether next character that will be transmitted is an ‘b’ or ‘q’).

• A symbol frequency table is required at compression time as well as decompression time. So it is necessary to transmit the table along with the encoded data. Various implementations of the Huffman Algorithm use different ways to store as well transmit the table.

• One of the alternatives is to use a universal symbol frequency table. This has an advantage that it may not be stored/transmitted along with Huffman codes as this
table is known in advance to parties involved in the process. A risk involved in this approach is that in a certain situation input stream may totally mismatch the universal frequency table, (e.g. a text message in which there are more ‘z’ than ‘a’ characters). In this situation we would not obtain a very good compression ratio.

- While compressing a file using static Huffman encoding, we need to do two passes on the file: in the first pass we make the Symbol Frequency table and in the second pass we compress the data.

- For files containing a large number of characters, building a symbol frequency table consumes a lot of time as which eventually increases the access time.
CHAPTER 3
INTRODUCTION TO CLASS TA

The Internet Web has evolved as an essential part of our daily life. Our growing dependence on it concludes it to be a necessity and hence there is a constant increase in our expectations. Owing to this, within the last few years Internet and Web technology has grown from just being a way to exchange information to a platform of interactive processes. In the beginning the Web was used only to publish information which the user would read. This period lasted from 1990 to early 2000 and was referred to as Web 1.0. To overcome the drawbacks of Web 1.0, developers came up with Web 2.0 which is an interactive version of Web 1.0 where users could participate and converse. Web browsing became collaborative as well as user friendly. Developers wanted to give people the same experience as desktop applications so that users may not get information just from the Web but also perform certain actions on the Web. Progressively, the term ‘Web’ was taken over by ‘Web application’. Today’s sophisticated Web applications can easily compete with desktop applications and are usually referred to Web 2.0 applications.

With the increasing demands of dynamic Web applications, the inclination of the developers towards newly emerging technologies has increased. AJAX deals with exchanging data with a server, and updating parts or sections of a Web page without reloading the whole page. AJAX programming involves a number of technologies, such as XHTML, CSS, Document Object Model (DOM), XML, XSLT, XMLHttpRequest object and JavaScript. AJAX and Web 2.0 together present the progression of Internet standards towards a better application development platform.

At San Diego State University, AJAX has been used to develop a Course-Management System (CMS) named ClassTA which is meant to be used by educators and students worldwide [38]. ClassTA is a combination of several subsystems. Some of its subsystems are QuizBuilder, Roster Management and Help system. The AJAX programming methodology has been used in the development of ClassTA as well as most of its subsystems [39]. Stating the example of one of its subsystem, The Help System is made dynamic through
the use of AJAX-style development. The Help system is enriched with a “wiki” which in turn is a web page or part of a page that allows users to add or modify the content [40]. The use of a “wiki” inside the Help System gives the system an interactive environment. Along with the use of AJAX as the main development technology, the idea of embedding a “wiki” in the Help System justifies the system as a Web 2.0 application.

3.1 BACKGROUND

ClassTA is a new kind of Web-based teaching assistant. It is meant to be used by teachers and students of grade school as well as graduate school. ClassTA is a web application which, apart from being intuitive and interactive, is rich in content and functionality. The ClassTA Web application provides the look and feel of a desktop application. Unlike the other CMS systems which utilize menu driven interfaces, ClassTA has followed the unique approach of designing a desktop like interface. The interface is designed to model and resemble the actual physical desk of an instructor or student, complete with files, folders, and storage devices.

In ClassTA, there are several functionalities and each one of them has a separate interface which is known as a ClassTA desktop. These desktops resemble the window of a contemporary operating system in looks and feel. Figure 3.1 shows the example of the interface when the user logs into ClassTA. In terms of vocabulary, they are known as a user Desktop. The Desktop contains several icons just like the icons inside a window of a Microsoft Windows® operating system Explorer window. When any of the icons on the Desktop is clicked, the respective desktop opens up in a different interface which resembles a browser tab such as that of Windows Internet Explorer® or Mozilla Firefox®. For example, Figure 3.2 shows the state of ClassTA after clicking on the icon titled ‘Test-101’ from Figure 3.1: a tab named ‘Test-101’ opens up. The prior shown example shows an interface which perfectly depicts the idea of how ClassTA has the look and feel of a desktop application. ClassTA provides speedier responses to user actions. Except for the initial loading delay, all other responses are instantaneous. For example, when a user clicks on any icon in the Desktop, shown in Figure 3.1, the respective tab opens instantly. Looking at the layout of traditional websites we can see that the structure is made up of a collection of Web pages which are linked to each other via links and when we traverse from one page to
Figure 3.1. ClassTA desktop.

Figure 3.2. Desk for test-101.
another, it takes time for a page to load and appear in a browser. Contradictory to this, ClassTA takes a different approach. Instead of loading the whole page it just loads a section of it thereby reducing the delay shown in communicating to the server.

In order to execute JavaScript on client’s machine, the JavaScript has to come from some resource initially and needs to be copied to a client’s machine. Small JavaScript codes are usually inserted inside an HTML head element and come directly with the webpage when a user initially loads a page. However, if the Web page is heavily based on JavaScript (which is the case in AJAX architecture), it is not an efficient practice to keep all required JavaScript in HTML directly. Instead, web developers put references for the scripts stored somewhere on the server in their HTML Web page and when the pages get loaded for the first time, these scripts are downloaded with a transport protocol (HTTP most of the time) and populated in the head element inside the HTML page. Also, there are ways in which a user can cache these scripts, so that if a user frequently accesses a Web page in some permissible interval of time, the page does not need to be loaded again from the server. As a result, the resources required to perform this operation can be saved to achieve a performance gain. However, in order to display a page, all these scripts may not be required to be loaded in the first pass (ideally it should not) and only a few of them need to be used. Also as these scripts are getting downloaded with sequential HTTP calls, it is again consumption of resources that are not required; as network, processor, memory and browser can handle higher load then loading each of them one by one.

3.2 USE OF HUFFMAN ENCODING IN CLASSTA

For this thesis, Huffman coding has been used to encode the existing ClassTA rich Internet application. All the JavaScript files are encoded using Huffman coding and then converted to base 64 format before storing on the server. When a request is made for the pages, those respective pages are fetched from the server and are decoded in the client browser using Huffman coding and loaded at the client side. This use of Huffman coding saves the storage space as well as reduces the time delay for loading pages on the client side.
CHAPTER 4

SYSTEM DESIGN AND IMPLEMENTATION

This chapter contains a comprehensive description of the JavaScript Huffman encoding system using UML diagrams and narrative text. The whole system is divided into two subsystems:

- The Huffman Encoder subsystem
- The Decoding subsystem

This chapter describes the detailed software design and implementation details for the two subsystems and the interaction between them. There were goals to be achieved through the design and development of this system. They are as follows,

- Compression of raw JavaScript files
- Decompression of compressed files at the client side
- Reduce load time of ClassTA
- Comparison of load times of various systems having JavaScript compression

4.1 CLASS DIAGRAM

Class diagrams are best used to represent object models or object oriented architecture. Class diagrams depict the classes for a system and the interrelationships between those classes.

Figure 4.1 shows the class diagram of the Huffman Encoder developed as part of this thesis. This class diagram is meant for the Huffman Encoding sub system,. The main classes here are the Huffman, Huffman Tree, Priority Queue and frequency table entry. Huffman uses other classes such as Base 64, Clear Data and Frequency table entry. Huffman Tree is a part of Huffman and hence there is a derivation relationship shown between them. Huffman tree in turn uses a priority queue. Huffman tree is a kind of binary tree which is depicted by the generalization relationship between them.

Figure 4.2 shows the class diagram of Decoding. This class diagram is meant for the second subsystem, Decoding. The main classes here are the Main_Script, FileLoader,
Figure 4.1. Class diagram of Huffman Encoder.
Figure 4.2. Class diagram of Decoder.
Decode_Javascript and JavascriptManager. Core_Functions are support functions for Decode_Javascript class. HuffmanTree and Node class are part of Extract_Data class.

4.2 USE CASE DIAGRAM

A use case diagram is a set of scenarios that describes an interaction between a user and a system. A use case diagram displays how the actors interact or participates in the use cases. The two main components of a use case diagram are use cases and actors. An actor represents a user or another system that will interact with the present system. A use case is an external view of the system that represents some action or a process the user might participate-in in order to complete a task.

Figure 4.3 shows two kinds of actors in our system, namely User and Admin. The user participates in the scenario where a browser is used for opening a website and the Admin is responsible for the process of encoding of the JavaScripts. The encoding can be done in two ways: it can either be done for a specific JavaScript or could be done for all contents of the Web page.

4.3 SEQUENCE DIAGRAM

A sequence diagram portrays a scenario that shows object interactions in a time-based sequence depicting the occurrence of events. Sequence diagrams show the interaction among the objects of classes and help provide essential information to determine class responsibilities and interfaces.

Figure 4.4 shows interaction between six different class objects. Various events are shown along the time line. These events give an idea of the sequence that the events follow in the context of occurrence. These events also help in depicting which objects interact in a particular time frame of reference. For an example, ‘inquires database’ and ‘encode all’ are the events which occurs between the objects of the class Admin and Database and we can also conclude that the event ‘encode all’ can take place only after ‘inquires database’ takes place.

4.4 DATA FLOW DIAGRAM

Data flow diagrams can be used to provide a clear representation of any process or function of a system. The series of diagrams starts with a general picture of a single process
Figure 4.3. Use case diagram.
Figure 4.4. Sequence diagram.
and continues by analysing each of the functional areas of interest. This analysis can be continued precisely up to the level of detail that is required. A top-down approach is adopted to conduct the analysis in a targeted way. The result is a series of diagrams that helps us clearly understand the business processes of the entire system. A business model comprises one or more data flow diagrams.

Initially a context level diagram is drawn, which is a simple representation of the entire system under consideration. This is followed by a level 1 diagram; which provides an overview of the major functional areas of the business. Any of the functionality in Level -1 can be analyzed further, giving rise to a corresponding level 2 business process diagram. This process of more detailed analysis can then continued through level 3, 4 and so on.

**4.4.1 Context Level**

Figure 4.5 is a context level diagram where in the whole system is depicted as a single process. The initial input and the final output of the process are shown. In this sub system, a Huffman object takes raw JavaScript files and converts it into Huffman encoded JavaScript and further converts it into Base64 format which is the output of this subsystem.

![Figure 4.5. Context (Level – 0 Data Flow ) diagram.](image)

**4.4.2 Level–1 Data Flow Diagram**

Figure 4.6 shows Level-1 data flow diagram. The level 1 diagram breaks down the entire single process into three parts, namely Huffman Encoder, Storage, and Decoding. The Huffman Encoder process takes JavaScript as an input and converts it into encoded Base64
Figure 4.6. Level-1 data flow diagram.

format. The intermediate output from the Storage process is stored in a ClassTA database. That in turn is given as an input to the decoding process which converts it into the original JavaScript.

4.4.3 Level – 2 Data Flow Diagram

Level-2 of DFD diagrams are decomposition of two of the process shown in level-1 diagrams. There should be a level-2 diagram for each and every process shown in the level-1 diagram.

Figure 4.7 shows Level-2 Data flow diagram. The Level-2 Diagram is an expansion of one of the processes of Level-1. The process that is being considered here is ‘Huffman
Figure 4.7. Level-2 Data flow diagram for encoding.

Encoder’. Level-1 process is expanded to four other processes. A level of consistency is maintained in terms of initial input and output while converting from Level-1 to Level-2.

Figure 4.8 shows a detailed DFD of the Decoding process that is displayed in a Level -1 DFD. It has been divided to six other processes. They give a detailed description of the functional units of decoding.

4.5 DATA DICTIONARY

A data dictionary is a compilation of descriptions of the data objects in a data model which could be used as a reference whenever required and refers to the database schema and the data structures used to store the data dictionary. A dictionary consists of a detailed description of all the fields of a database table. Along with the name and data type, a data dictionary also describes the constraint pertaining to that field.
Figure 4.8. Level-2 Data flow diagram for decoding previously encoded JavaScript.
Table 4.1 shows script table that contains raw JavaScript files. Table name: Scripts
Description: Contains raw JavaScript files.

### Table 4.1. Scripts

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Attribute</th>
<th>Constraints</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRL</td>
<td>bigint(32)</td>
<td>AUTO_INCREMENT</td>
<td>Primary Key</td>
<td>Primary Key</td>
<td></td>
</tr>
<tr>
<td>filename</td>
<td>varchar(64)</td>
<td></td>
<td></td>
<td>Raw JavaScript filenames</td>
<td></td>
</tr>
<tr>
<td>fullpath</td>
<td>text</td>
<td></td>
<td></td>
<td>Path in the project</td>
<td></td>
</tr>
<tr>
<td>javascript</td>
<td>longblob</td>
<td>BINARY</td>
<td></td>
<td>NULL</td>
<td>Raw files</td>
</tr>
<tr>
<td>groupname</td>
<td>varchar(32)</td>
<td>Secondary Key</td>
<td>unspecified</td>
<td>Group name where files belongs to</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>int(10)</td>
<td></td>
<td>0</td>
<td>Updated version number</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>char(1)</td>
<td></td>
<td></td>
<td>Current state</td>
<td></td>
</tr>
<tr>
<td>timestamp</td>
<td>timestamp</td>
<td>ON UPDATE CURRENT_TIMESTAMP</td>
<td>CURRENT_TIMESTAMP</td>
<td>Current time stamp</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 shows groups table that contains raw JavaScript filenames within specific group. Table name: groups. Description: Contains raw JavaScript file names within specific group.

Table 4.3 shows huffman table that contains JavaScript files encoded into Base64 format with specific groups. Table name: Huffman. Description: Contains JavaScript files encoded into Base64 format with specific groups.
### Table 4.2. Groups

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Attribute</th>
<th>Constraints</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>iRL</td>
<td>int(32)</td>
<td>AUTO_INCREMENT</td>
<td>Primary Key</td>
<td>Primary</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Key</td>
<td></td>
</tr>
<tr>
<td>groupname</td>
<td>Varchar(64)</td>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td>Specific group name</td>
</tr>
<tr>
<td>files</td>
<td>text</td>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td>Raw JavaScript file names</td>
</tr>
</tbody>
</table>

### Table 4.3. Huffman

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Attribute</th>
<th>Constraints</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupname</td>
<td>varchar(64)</td>
<td></td>
<td>Primary Key</td>
<td></td>
<td>Specific group name</td>
</tr>
<tr>
<td>base64</td>
<td>longtext</td>
<td></td>
<td></td>
<td></td>
<td>Encoded JavaScript files in base64 format</td>
</tr>
<tr>
<td>timestamp</td>
<td>timestamp</td>
<td>ON UPDATE CURRENT_TIMESTAMP</td>
<td></td>
<td>CURRENT_TIMESTAMP</td>
<td>Current time stamp</td>
</tr>
</tbody>
</table>
CHAPTER 5

EXPERIMENTS AND RESULTS

This chapter gives an overview of the experiments conducted. It briefly explains the environment setup and the tools used for the experiments. It also captures the results of the experiment.

5.1 GOOGLE CHROME DEVELOPER TOOLS

Google Chrome is open source browser. Its rendering engine was developed with the help of WebKit. WebKit is an open source web browser engine. WebKit is also the name of the Mac OS X system framework version of the engine that's used by Safari, Dashboard, Mail, and many other OS X applications [41]. Chrome provides tools which helps developer for Web developments. The following tools were used during development and testing:

- Web Inspector: This allows one to take a closer look at any element on the currently open page. It is available by right clicking on an element. The Web Inspector allows you to browse page elements and view object properties and style. This is a feature from the WebKit base.

- JavaScript console: This allows you to enter command-line JavaScript code that can access page elements. It opens within the Web Inspector window located in the bottom portion of the screen.

- JavaScript debugger: A rudimentary command line JavaScript debugger.

- Task Manager: This allows you to view the current processes running within Chrome; it is analogous to the Task Manager available in Windows. It shows the system resources by a process. This includes memory, network, and CPU usage. A button is provided to end a process along with link to a report that breaks down memory usage for individual processes.

5.2 NETWORK PANEL

The Network panel lets one inspect resources that are downloaded over the network [42].

Follow the steps below to explore the Network panel:

- Open Google Chrome
Open Developer tools window
Select Network panel

Figure 5.1 shows the Network panel during testing of the ClassTA Huffman version. This shows a timeline waterfall of network activities. Latency, the time between making the request and the server's first response, is shown in the lighter shade within each bar. The blue and red vertical lines - on the right-hand side in the below screenshot - indicate firing of the DOMContentLoaded and Load events respectively. This helps clarify the time it takes for pages to load and helps one improve website load times. One can choose to view all resources or a subset using the All, Documents, Stylesheets, Images, and other scope bars below the panel selection icons. Figure 5.2 shows web inspector.

Figure 5.1. Task manager.

The DOMContentLoaded event is one of the main pillars of modern, unobtrusive JavaScript usage. This event fires after the HTML code has been fully retrieved from the server, the complete DOM tree has been created and scripts have access to all elements via the DOM API. Figure 5.3 shows network panel of Chrome developer tool.

5.3 EXPERIMENT SETUP

We set up the experiment that uses page load time as an indicator to measure the efficiency of the algorithm. There are four methods used, (1) static, (2) dynamic loading, (3) Shrink Safe, and (4) Huffman encoding. The experiment is performed over five different networks, (1) 3mbps, (2) 2.03 mbps, (3) 6 mbps, (4) 19.47 mbps, and (5) 100 mbps. The setup is delineated as:
Figure 5.2. Web inspector.

Figure 5.3. Network panel of Chrome developer tools.
• The Google Chrome browser (version: 11.0.696.68) is used to load the four different Universal Resource Locators (URLs). We used Chrome developer tools to measure the page load time accurately.

• Each URL is loaded in the browser using each method and the page load time is measured. As a precondition the browser cache and cookies are cleared to make sure that the browser does not fetch data locally.

• Matlab software (version: 2010b) is used to plot the results in graphical manner. The graphs are generated using least square feet with smooth fit lines approach. There are three types of graphs generated for each algorithm over each network downlink speed,
  • The page load time taken per URL.
  • The combined page load time of all URLs.
  • The average page load time over ten iterations.

The graphs are illustrated in next section with detailed analysis.

5.4 RESULTS GRAPH

Below mentioned is the result summary shown in the plotted graphs. These graphs show the page load time for several networks with different bandwidths.

5.4.1 3.0 Mbps Network

Plotted graph in Figure 5.4 shows the load time results obtained over the number of trials for the 3.0Mbps Network using the Huffman Encoding approach. From the obtained results, the upper peak, average and lower peak load time values are – 8.76 s, 7.94 s and 6.99 s respectively.

Plotted graph in Figure 5.5 shows the load time results obtained over the number of trials for the 3.0Mbps Network using the Shrink Safe approach. From the obtained results, the upper peak, average and lower peak load time values are – 8.75 s, 7.78 s and 6.35 s respectively.

Plotted graph in Figure 5.6 shows the load time results obtained over the number of trials for the 3.0Mbps Network using the Dynamic ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 12.81 s, 10.78 s and 8.18 s respectively.

Plotted graph in Figure 5.7 shows the load time results obtained over the number of trials for the 3.0Mbps Network using the Static ClassTA approach. From the obtained results,
Figure 5.4. 3.0 Mbps network for ClassTA’s Huffman version.

Figure 5.5. 3.0 Mbps network for ClassTA’s Shrink Safe version.
the upper peak, average and lower peak load time values are – 14.24 s, 10.44 s and 8.59 s respectively.

Plotted graph in Figure 5.8 shows the load time results obtained over the number of trials for the 3.0Mbps Network using the four different approaches. The graph is plotted using least squared fit approach with smooth fit lines instead of raw data results obtained. From the obtained results, average load time results for Huffman Encoding and Shrink Safe approach are better than the average load time results for Dynamic ClassTA and Static ClassTA.
5.4.2 6.0 Mbps Network

Plotted graph in Figure 5.9 shows the load time results obtained over the number of trials for the 6.0Mbps Network using the Huffman Encoding approach. From the obtained results, the upper peak, average and lower peak load time values are – 4.08 s, 3.56 s and 2.54 s respectively.

Plotted graph in Figure 5.10 shows the load time results obtained over the number of trials for the 6.0Mbps Network using the Shrink Safe approach. From the obtained results, the upper peak, average and lower peak load time values are – 4.51 s, 3.53 s and 2.61 s respectively.

Plotted graph in Figure 5.11 shows the load time results obtained over the number of trials for the 6.0Mbps Network using the Dynamic ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 6.43 s, 5.63 s and 4.08 s respectively.

Plotted graph in Figure 5.12 shows the load time results obtained over the number of trials for the 6.0Mbps Network using the Static ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 6.51 s, 5.28 s and 4.44 s respectively.
Figure 5.9. 6.0 Mbps network for ClassTA’s Huffman version.

Figure 5.10. 6.0 Mbps network for ClassTA’s Shrink Safe version.
Figure 5.11. 6.0 Mbps network for ClassTA’s dynamic version.

Figure 5.12. 6.0 Mbps network for ClassTA’s static version.

Plotted graph in Figure 5.13 shows the load time results obtained over the number of trials for the 6.0Mbps Network using the four different approaches. The graph is plotted using least squared fit approach with smooth fit lines instead of raw data results obtained. From the obtained results, average load time results for Huffman Encoding and Shrink Safe approach are better than the average load time results for Dynamic ClassTA and Static ClassTA.
5.4.3 2.03 Mbps Network

Plotted graph in Figure 5.14 shows the load time results obtained over the number of trials for the 2.03Mbps Network using the Huffman Encoding approach. From the obtained results, the upper peak, average and lower peak load time values are – 8.29 s, 7.36 s and 6.70 s respectively.

Plotted graph in Figure 5.15 shows the load time results obtained over the number of trials for the 2.03Mbps Network using the Shrink Safe approach. From the obtained results, the upper peak, average and lower peak load time values are – 8.90 s, 7.43 s and 6.36 s respectively.

Plotted graph in Figure 5.16 shows the load time results obtained over the number of trials for the 2.03Mbps Network using the Dynamic ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 9.84 s, 9.00 s and 8.55 s respectively.

Plotted graph in Figure 5.17 shows the load time results obtained over the number of trials for the 2.03Mbps Network using the Static ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 8.90 s, 8.09 s and 7.23 s respectively.
Figure 5.14. 2.03 Mbps network for ClassTA’s Huffman version.

Figure 5.15. 2.03 Mbps network for ClassTA’s Shrink Safe version.
Figure 5.16. 2.03 Mbps network for ClassTA’s dynamic version.

Figure 5.17. 2.03 Mbps network for ClassTA’s static version.

Plotted graph in Figure 5.18 shows the load time results obtained over the number of trials for the 2.03Mbps Network using the four different approaches. The graph is plotted using least squared fit approach with smooth fit lines instead of raw data results obtained. From the obtained results, average load time results for Huffman Encoding and Shrink Safe approach are better than the average load time results for Dynamic ClassTA and Static ClassTA.
5.4.4 19.47 Mbps Network

Plotted graph in Figure 5.19 shows the load time results obtained over the number of trials for the 19.47Mbps Network using the Huffman Encoding approach. From the obtained results, the upper peak, average and lower peak load time values are – 3.68 s, 2.95 s and 2.44 s respectively.
Plotted graph in Figure 5.20 shows the load time results obtained over the number of trials for the 19.47Mbps Network using the Shrink Safe approach. From the obtained results, the upper peak, average and lower peak load time values are – 3.81 s, 2.89 s and 1.93 s respectively.

![Figure 5.20. 19.47 Mbps network for ClassTA’s Shrink Safe version.](image)

Plotted graph in Figure 5.21 shows the load time results obtained over the number of trials for the 19.47Mbps Network using the Dynamic ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 5.24 s, 4.35 s and 3.23 s respectively.

Plotted graph in Figure 5.22 shows the load time results obtained over the number of trials for the 19.47Mbps Network using the Static ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 4.93 s, 4.00 s and 3.11 s respectively.

Plotted graph in Figure 5.23 shows the load time results obtained over the number of trials for the 19.47Mbps Network using the four different approaches. The graph is plotted using least squared fit approach with smooth fit lines instead of raw data results obtained. From the obtained results, average load time results for Huffman Encoding and Shrink Safe approach are better than the average load time results for Dynamic ClassTA and Static ClassTA.
Figure 5.21. 19.47 Mbps network for ClassTA’s dynamic version.

Figure 5.22. 19.47 Mbps network for ClassTA’s static version.
Figure 5.23. 19.47 Mbps network for ClassTA’s loadtime.

5.4.5 100 Mbps Network

Plotted graph in Figure 5.24 shows the load time results obtained over the number of trials for the 100.0Mbps Network using the Huffman Encoding approach. From the obtained results, the upper peak, average and lower peak load time values are – 2.76 s, 2.24 s and 1.71 s respectively.

Figure 5.24. 100 Mbps network for ClassTA’s Huffman version.
Plotted graph in Figure 5.25 shows the load time results obtained over the number of trials for the 100.0Mbps Network using the Shrink Safe approach. From the obtained results, the upper peak, average and lower peak load time values are – 2.87 s, 2.31 s and 1.77 s respectively.

Figure 5.25. 100 Mbps network for ClassTA’s Shrink Safe version.

Plotted graph in Figure 5.26 shows the load time results obtained over the number of trials for the 100.0Mbps Network using the Dynamic ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 3.48 s, 2.89 s and 2.36 s respectively.

Plotted graph in Figure 5.27 shows the load time results obtained over the number of trials for the 100.0Mbps Network using the Static ClassTA approach. From the obtained results, the upper peak, average and lower peak load time values are – 3.19 s, 2.70 s and 2.36 s respectively.

Plotted graph in Figure 5.28 shows the load time results obtained over the number of trials for the 100.0Mbps Network using the four different approaches. The graph is plotted using least squared fit approach with smooth fit lines instead of raw data results obtained. From the obtained results, average load time results for Huffman Encoding and Shrink Safe approach are better than the average load time results for Dynamic ClassTA and Static ClassTA.
Figure 5.26. 100 Mbps network for ClassTA’s dynamic version.

Figure 5.27. 100 Mbps network for ClassTA’s static version.
5.4.6 Data Compression During Page Load of ClassTA

Plotted graph in Figure 5.30 shows the compress Data (X-axis) over different types of method (Y-axis). From the obtained results, Huffman Encoding is more efficient compare to other approaches.
Figure 5.29. Average loadtime of ClassTA on various networks.
Figure 5.30. Data size during page load of ClassTA.
CHAPTER 6

CONCLUSION

In this thesis, we show one of the many significant usecases of Huffman Encoding technique. We propose a novel and efficient runtime JavaScript decompression technique using Huffman Encoding. We have studied various technologies such as CSS, JSON framework, AJAX that assist in improving performance of Web page based mechanism as well as increasing the elegance in its appearance. Since AJAX allows Web pages to be updated asynchronously by exchanging small amount of data with the server, AJAX helps make pages render smoothly. Google Maps, Gmail, Yahoo, LinkedIn could be considered as best examples of using Ajax technologies.

There are two main categories of compression: lossy and lossless. Since lossy compression produces smaller file than any known lossless techniques, they are relatively advantageous in audio and still image areas. However, even a single bit error using a lossy scheme would result in a JavaScript load failure, we cannot employ lossy compression. Huffman falls into the lossless compression category which is a statistical technique that attempts to minimize the number of bits used for representing a string of characters or symbols. Huffman allows the encoded symbols to vary in length.

The ClassTA is a Web portal that contains a collection of JavaScript files. Huffman encoding is used to compress the files at the server side. At client side, the runtime decompression is performed using four approaches, (1) static, (2) dynamic loading, (3) Shrink Safe, and (4) Huffman encoding. Shrink Safe eliminates unnecessary white spaces, newlines, CRs, and comments to help reduce the file size. Shrink Safe is industry known tool used for file compression.

We have illustrated architectural design, corresponding use case, call flow, and data flow diagrams. From the results of the experiments performed, we show that our approach produces optimum or equal best load time compared to the approaches mentioned earlier. At each downlink speed network, Huffman encoding performs much better than both dynamic and static approaches. At both 2.03Mbps and 100Mbps, Huffman performs even better than
Shrink Safe. The compression of Huffman encoding is optimum among all the approaches mentioned in Figure 5.30 (p. 65).
REFERENCES


