ADAPTIVE CDCLZW ALGORITHM FOR DATA COMPRESSION

Xiaoqin LIU, Jianxin LIAO, Qiwei SHEN

State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications, Beijing 100876, P. R. China
liuxiaoqin@ebupt.com, liaojianxin@ebupt.com, shenqinwei@ebupt.com

Abstract: With the development of technology, billions of clients visit web servers to surfing on the Internet every day. Insight is a code flow tracing project for web servers. The tracing data in Insight rising with the increasing quantity of web accessing is enormous. JavaScript Object Notation (JSON), a lightweight, text-based, language-independent data interchange format, is used in Insight to transfer data between clients and server. This procedure often contains the transferred code flows with most of data is similar, which can be compressed to save bandwidth. This paper proposes a new adaptive Content Defined Chunking Lempel Ziv Welch (CDCLZW) algorithm for the Insight, to compress JSON data as far as possible and save the network bandwidth. . The CDCLZW algorithm uses Template Library and Rabin's fingerprints to find same data and avoid transferring them between clients and servers. The test environment is developed to investigate the new algorithm, and the results demonstrate that CDCLZW can insure the compression ratio steady and performs better than GNUzip (gzip) and bzip2 algorithms.

Keywords: CDC; LZW; JSON; Compression; Insight; Code flow

1 Introduction

The mobile Internet combines the mobile communications and the Internet together. Accessing Internet is no longer just by desktop alongside with the expansion of mobile networks. With the large number of users, mobile Internet is considered owning the largest market potential. Many developers devote themselves to mobile applications. To reduce the development time many excellent frameworks are put forward. These frameworks hide the details and just provide interfaces to programmers so that troubleshooting errors becomes harder. To solve these problems and make agile development, Insight Project is proposed.

Insight Project is based on AspectJ to trace code flow. Insight Project is mainly consisted of three parts as shown in Figure 1. One is client-plugin, located in web service server Like Tomcat, using various plugins to trace and record web service flow as shown in Figure 2. After completing gathering information about a specific web service code flow, client-plugin sends this information in JSON format to center transfer servers. Once center transfer servers receive these data, they do analysis and grouping then send processed data to center server for storage and processing and displaying. As web service server is always busy and client-plugin would send thousands of data per second. In a web server, the accessible URL (Uniform Resource Locator) is limited, so is code flow. So in most cases, transferred data is very similar and many redundant data waste bandwidth.

Reducing the quantity of transferring data and simplifying the process of parsing data is more and more important thus JavaScript Object Notation (JSON) is proposed.

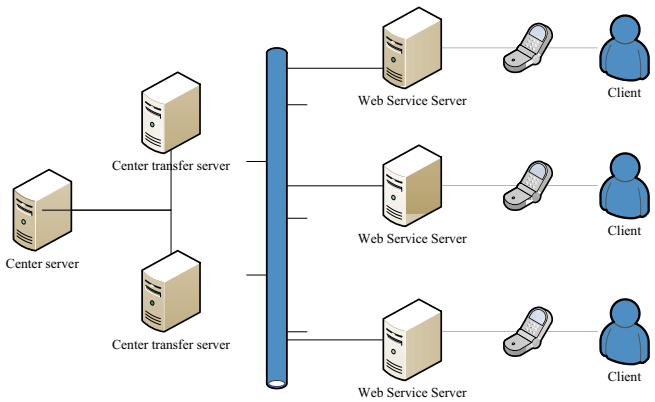

Figure 1 Architecture of insight project

978-1-4799-0094-7/13/$31.00 ©2013 IEEE

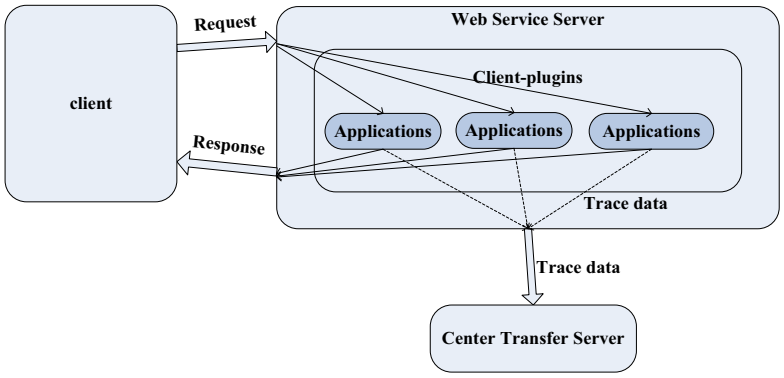

Figure 2 Client plugin deployment

JSON is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript (European Computer Manufactures Association Script) Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data. As JSON is simple and efficient, many applications use it to transfer data between clients and servers [1].

Real-time data transmission is of great importance in Insight. Client-plugins snatch network resources to send tracing data. Data transfers saturate bottleneck links and cause unacceptable delay. Currently, data compression is commonly used to save bandwidth. LZW algorithm proposed in Welch's 1984 paper [2] is one of the text compression algorithm and it's simple to implement, and has the potential for very high throughput in both hardware implementations and software implementations. In LZW, 8-bit data is encoded as fixed-length 12-bit codes. The codes from 0 to 255 represent 1-character sequences consisting of the corresponding 8-bit character, and the codes 256 through 4095 are created in a dictionary for sequences encountered in the data as it is encoded. At each stage in compression, input bytes are gathered into a sequence until the next character would make a sequence for which there is no code yet in the dictionary. The code for the sequence (without that character) is added to the output, and a new code (for the sequence with that character) is added to the dictionary [3].

While in Insight, similar data is in majority, exploiting the similarity between data could reduce the cost of transmission and simplify the storage management. Duplicate checking technology is based on dividing Data Object (DO) [4] into consecutive, non-overlapping chunks and computing hash value of each chunk as the identifier of the chunk (chunkId). An object may be divided into either fixed-size or variable-sized blocks. The fixed block-size strategy is fixed-sized chunking (FSC), which is the simplest duplicate checking strategy. FSC is very efficient and convenient, but is highly sensitive to the sequence of edits and modifications performed on consecutive versions of an object [4]. To solve problems with the FSC technique, content defined chunking (CDC) [5] is proposed. CDC partitions objects into variable-size blocks or chunks. CDC employs Rabin's fingerprints [6] to choose partition points. Fingerprints help CDC to identify the relative points at which the object was partitioned in previous versions without maintaining any state information [5].

This article based on LZW and CDC algorithm, proposes a new algorithm to compress data for Insight Project. The rest of this paper is organized as follows. Section 2 describes the related works, Section 3 describes CDCLZW algorithm in detail and provides an algorithm implementation with pseudo-code. Experiment results are given in Section 4. Finally, conclusions are dedicated in Section 5.

2 Related work

Currently numerous researches have been proposed based on different text compression schemes like string matching [7] and statistics model. Letter frequency is one of the most popular schemes pioneered by Huffman. While fixed-length coding is generally used in many text compression algorithms, algorithms based on statistics variable-length coding is much more popular. Various length of code is specified according to various distribution probabilities of source symbols. In this way, the average code length is shortest in company with highest encoding efficiency [8-9]. Shannon–Fano coding is the earliest variable-length coding [10]. Huffman coding has many commons with Shannon-Fano, but it is much more efficient and has been used much more widely. Statics coding is quite used in image compression and text compression.

Although coding based on statistics has many benefits, setting up an accurate statistics model is very complicated and troublesome. So the other scheme based on dictionary coding is proposed and widely used in text compression, audio compression and image compression. It maintains a dictionary which contains many strings and corresponding codes. Lempel-Ziv 1977 compression format (LZ77) and Lempel-Ziv 1978 compression format (LZ78) is the representative coding of this scheme. According to the store procedure is static or not, this coding procedure is dived into non-adaptive or adaptive. Based on LZ77 and LZ78, many improved algorithms are proposed like LZW.

In order to identify similar documents in a large collection of documents, fingerprints are proposed. As Rabin's fingerprints are very easy to compute over a sliding window, it is used extensively among fingerprinting techniques.

Identical data detection techniques are used to eliminate duplicate to reduce the cost of storage. Identical data includes identical file and identical data block. Whole file detection (WFD) checks duplicate content by using a unit in files [11]. Detecting duplicate data block techniques mainly includes FSC, CDC and sliding block techniques [12].

3 CDCLZW algorithm

In this section, CDCLZW algorithm is descripted in detail. The algorithm includes two parts: compression and decompression.

3.1 Compression

The process of compression is adaptive and applied in client-plugins, which consists of two parts, one is CDC based Content Eliminate, proposed in this article, and the other is LZW compression.

Content Eliminate, as shown in Algorithm 1, based on the content division of CDC algorithm, assumes *exp_chunk_size* and *magic_value* parameters and depending on its fingerprint value calculates chunkMask. It initializes a fixed-size window (generally 12 bytes to 48 bytes). Started from the beginning of DO, sliding the window by one byte per time and compute Rabin's fingerprint of the content within window and check whether fingerprint value satisfies boundary condition. If boundary condition is satisfied, compute the MD5 (Message Digest Algorithm) [13] value of content as *chunkId*. Through chunkId determines whether current chunk is in Template Library. If *chunkId* is already in Template Library, replace chunk content using a tuple <*offset*, *length*, *flag*, *chunkId*> (*flag* indicates the fourth element is a *chunkId* or chunk content.); or add a pair <*chunkId*, chunk content> to Template Library and replace chunk content using a tuple<*offset*, *length*, *flag*, chunk content>.

The algorithm description hides the following details:

1) A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. And fingerprints can be computed easily on sliding window of bytes. Rabin's fingerprints can be seen as random distribution. By setting boundary condition can control the size of chunk [5].

Unfortunately, depending on Rabin's fingerprint, the size of chunk is in fluctuation and could lead to some pathological behavior [5]. In order to prevent the pathological behavior, the minimum and maximum chunk size are set, which are not revealed in Algorithm 1.

Algorithm 1: Content Eliminate

```
Input: String originalData
       Integer exp_chunk_size
Output: String reconstructedData
Procedure:
1: Hash template := empty
2: reconstrucedData := empty
3: chunkMask := calculateMask(exp_chunk_size)
4: foreach byte position X in originalData do
5:     window := subString (originalData,X,size)
6:     fp := fingerprint(window)
7:      if (fp & chunkMask = magic_value)
8:     then
9:           mark X
10:    end if
11: end for
12: mark last position in originalData
13: firstpos := 0
14: offset := 0
15: foreach byte position X that is marked do
16:    length := X - firstpos
17:    chunk := subString(f,firstpos,length)
18:    md5 := MD5(chunk)
19:    bool isId := FALSE
20:    if (template.contains(md5))
21:    then
22:          isId := TRUE
23:          reconstructedData+=
createString(offset,length,isId,md5)
24:    else
25:           template.put(md5,chunk)
26:          reconstructedData +=
createString(offset,length,isId,chunk)
27:    end if
28:    firstpos := X
29:    offset += X - firstpos
30: end for
31: return reconstructedData
```

2) The Template Library's size is fixed. As the growth of data transmission, the size of Template Library grows. When the size of Template Library becomes enormous, time for searching a specific chunkId becomes unacceptable. So in algorithm, the size of Template is set to be fixed. There is an issue about how to swap when the Template Library is full.

In the Template Library, each item is a triple as (*chunkId*, chunk content, *counter*). When an item is hit, the counter of it will set to zero, and other items' counter will plus one. Once the Template Library is full, the item with largest counter will swap out.

3) Because of the possibility of collision, there exists risk of using hash value as chunk identifier. If two chunks with different content were assigned a same hash value, then these two chunks would be replaced with the same chunkId. It will cause decompression error at server-end. But as research indicated, MD5 has a strong capacity of anti-collision so that the probability of collision is extremely low [14]. So using MD5 hash value as a chunk identifier is adoptable in practice.

4) Template Library is empty initially. After execution of Content Eliminate, in order to transfer data for decompression, the reconstructed data will be larger than the original. In order to help relieve this situation, after eliminating same data, LZW algorithm is used to compress reconstructed data.

3.2 Decompression

Decompression implementation is applied at server-end. The process of decompression is divided into two stages: LZW decompression and Content Recovery which is the reverse processing of Content Eliminate.

When server-end receives data, LZW decompression algorithm is used firstly. Then Content Recovery use flag to recover data. If flag is true, Content Recovery search the Template Library to find the relatively chunk content. Or, Content Recovery just get chunk content in received data and compute the MD5 value of the chunk to put into Template Library. Thus there is no need to transfer Template Library between client-plugins and server-end. The algorithm is shown in Algorithm 2.

Algorithm 2: Content Recovery
Input: String receivedData
Output: String data
Procedure:

```
1: Hash template := empty
2: data := empty
3: List chunks := evalJson(receivedData)
4: foreach chunk in chunks do
5:if chunk.flag = TRUE
6:then
7:        content := template.get(chunk_info)
8:else
9:        content := chunk_info
10:         md5 := MD5(content)
11:       template.put(md5, content)
12:end if
13:data += content
14: end for
15: return data
```

There are some details behind the algorithm. In particularly when the Template Library is full, in client-plugin, item with the least hit ratio is swap out. Considering synchronization, the following situation will happen.

There are two thread T1 and T2 are going to send data, and both of them contains a chunk with id *D*. While the Template Library is full, the Library uses the strategy mentioned before to swap an item out and add chunk with Id *D*. Assuming T1 send chunk *D* with original data while *T2* send chunk *D* with its id. At the server-end, *T2* arrives first and can't find an item with id *D* in server-end library. Until *T1* comes, chunk with id *D* can be add to the Library. In order to solve this problem, a queue is maintained at the server-end to save the unresolved data. Once Template Library is changed, the queue will be notified to resolve the data within it.

4 Experimental result

As Rabin's fingerprints affect the size of chunk and are vital to the performance of compression. To achieve the best performance, magic value is adjusted to get the possible largest compression ratio.

To test and compare the effects of different magic values, a real Insight Project is used. To simplify deployment and comparing data directly, the center transfer servers are omitted and the client-plugins send data directly to the center servers.

Two client-plugins were deployed on a same server, while center server was on a different server. These two servers have different capacity. The server of client-plugins had 2 Xeon CPUs and 8 GB memory, and the server of center server had 4 Xeon CPUs and 16 GB memory. In a second, 10 requests, generated by Apache Jmeter evenly, were sent to the client-plugins. The source data size is the number of transferred data between client-plugins and center servers, as shown in Table I. From Table I, when magic value is set to 255 bytes, that is , the boundary condition is to check the lower-order 8 bits of a region's fingerprint, the largest compression ratio is achieved.

Table I Compression ratio for different magic value

Magic value (Byte)	Source Data Size(KB)	Compression ratio
63	3043966	0. 723
127	3457323	0. 501
255	3457934	0. 368
511	3457807	0. 402
1023	3440330	0. 439

In the next experiment, magic value is set as 255 to compress transferred data. To test and compare the performance of CDCLZW algorithm to gzip algorithm and bzip2 algorithm, these three algorithms is applied in the same Insight Project. In Figures 2~3, request rate were increased from 10 RPS (request per second) to 80 RPS.

Figure 3 shows the compression rate of three algorithms: CDCLZW, gzip and bzip2. CDCLZW always has the best performance. As the growing of RPS, CDCLZW performance becomes very steady and keeps compression ratio at approx. 20%.

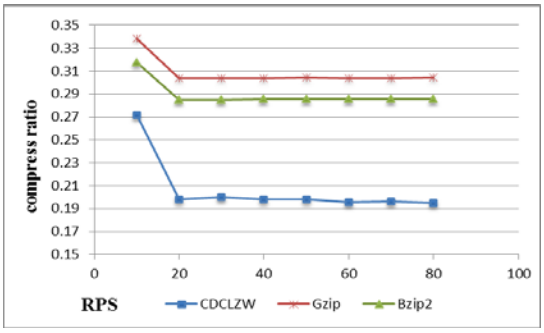

Figure 3 Compression ratio of CDCLZW, gzip and bzip2 algorithms

Figure 4 shows the average compression time per request of CDCLZW, gzip and bzip2 algorithms. It can be concluded that gzip has the least compression time but experienced several fluctuations. And CDCLZW compression time decreased as the increasing of RPS, and kept steady at approx. 11 MS (millisecond) per request.

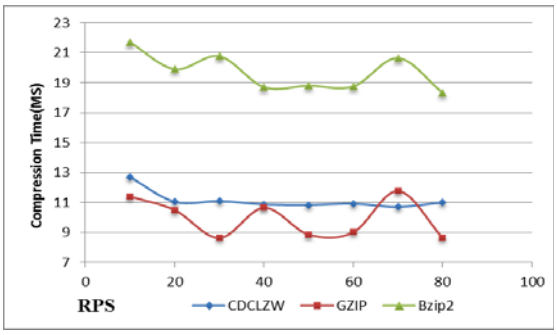

Figure 4 Compression time of CDCLZW, gzip and bzip2 algorithm

5 Conclusions

This paper proposes a new compression algorithm based on CDC and LZW to save bandwidth between client-plugins and center transfer servers. Applying this algorithm in Insight Project proves that its performance is good and steady. Along with the running of Insight Project, Template Library becomes more practical and more suitable to realistic requests. Then, compression ratio will benefit from this phenomenon.

However, as the growth of the Template Library, searching a chunk of it will be time-consuming. Conversely, if the size of Template Library is too small, the compression ratio will reduce. How to determine the size and how to organize the index of the Template Library is still a problem to effect the compression time of this algorithm and it will be compensated for by future work.

Acknowledgements

This work was jointly supported by the National Basic Research Program of China (No. 2013CB329100, 2013CB329102); National Natural Science Foundation of China (No. 61271019, 61101119, 61121001, 61072057, 60902051); PCSIRT (No. IRT1049).

References

[1] Nurseitov N, Paulson M, Reynolds R, Izurieta C: Comparison of JSON and XML Data Interchange Formats: A Case Study. CAINE 2009, 157-162.

[2] Welch, T.A. A Technique for High-Performance Data Compression. IEEE Computer 17,6 (June 1984), pp. 8-19.

[3] Horspool, R.N. Improving LZW. Proc. Data Compression Conference (DCC 91), Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, pp. 332-341.

[4] S. Quinlan and S. D. Venti: A New Approach to Archival Storage. In Proceedings of the USENIX Conference on File And Storage Technologies, January 2002.

[5] Bobbarjung D. R., Jagannathan S., AND Dubnicki C. Improving duplicate elimination in storage systems. Trans. Storage 2, 4 (2006), 424–448.

[6] Rabin, M. 1981. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University.

[7] M. Rodeh, V. R. Pratt and S. Even, Linear Algorithm for Data Compression via String Matching, J. of the ACM 28, 1 (January 1981), 16.

[8] L. Felician and A. Gentili, A nearly optimal Huffman technique in the microcomputer environment, Inf. Sys.12, 4 (1987), 371.

[9] R. G. Gallager, Variations on a Theme by Huffman, IEEE Trans. on Inf. Theory IT-24, 6 (1978), 668.

[10] J. B. Connell, 'A Huffman-Shannon-Fano code', Proc. IEEE, 61, (7), 1046–1047 (1973).

[11] W. J. Bolosky, S. Corbin, D. Goebel, J. R. Douceur.Single Instance Storage in Windows 2000. 4th UsenixWindows System Symposium, Aug 2000.

[12] Broder AZ. Identifying and filtering near-duplicate documents. In: Giancarlo R, Sankoff D, eds. Proc. of the 11th Annual Symp. on Combinatorial Pattern Matching. London: Springer-Verlag, 2000. 1−10.

[13] R.Rivest. The MD5 message-digest algorithm. RFC1321, Internet Engineering Task Force (1992).

[14] Black J. Compare-by-hash: a reasoned analysis. In Proceedings of the 2006 USENIX Annual Technical Conference. Boston, MA, USA: USENIX Association, 2006:85-90.