LZW Compression of interactive network traffic

Anders Klemets, SM0RGV

Swedish Institute of Computer Science
Box 1263
s-164 28 Kista
SWEDEN

ABSTRACT

This paper summarizes some aspects of data compression and describes how it can be used to increase the throughput of a computer communications link. The popular LZW algorithm is studied in detail, and special consideration is given to the problems that arise when using LZW to compress interactive traffic. An LZW implementation for the KA9Q Internet package for compression of interactive data is presented. Finally, the algorithm used in this implementation is compared to the V.42bis modem data compression standard

1. Introduction

There are different to ways increase the effective transfer speed, or throughput, of a computer communications link. An obvious way might be to increase the bit rate of the hardware. This will usually involve obtaining faster modems, or switching from an RS-232 interface to some other faster interface. But there might be factors that limit the extent to which the physical speed can be increased. For instance, all physical media have a limited bandwidth. For telephone lines the bandwidth is only about 3 kHz. For radio channels the bandwidth is primarily limited by regulations. The usefulness of the equipment may also be limited by its price and by the need for backwards compability.

In some cases one may achieve substantial improvement of throughput by replacing or modifying the different communications protocols that are in use. If the link level protocol has not been properly designed, it might impose considerable overhead. It has also been shown to be possible to make significant improvements to the performance of network and transport protocols. In some cases just altering the timing and flow control mechanisms is enough [Jacobs88].

It might actually be possible to improve the transfer rate by adding yet another protocol to the existing protocol stack. This paper describes how a data compression protocol can be used at the presentation layer.

2. Compression methods

Compression methods can be grouped into different categories. For instance, when compressing sound or images, it can make sense to discard some of the information. The decompressed data will not be identical to the original, but when done correctly it may not be noticeable to humans. Such compression is called irreversible. In the general case, however, one would like to have reversible compression, i.e. the decompressed data should be identical to the original data. Anything less would be unacceptable when, for example, transferring binary computer programs.

There are two important compression techniques: statistical coding, and dictionary coding. In statistical coding; each character is assigned a codeword based upon how frequently it occurs. Common characters get short codewords and the more unusual characters get longer codewords. In dictionary coding, strings of characters are entered in a dictionary. Whenever the string occurs, a codeword representing

its dictionary entry is used. See a compression textbook *such* as [Storer88] for a more comprehensive description of these techniques.

Dictionary ***coding*** is always less efficient than statistical coding. [Bell89] (It gives a lower compression ratio. Compression ratio is defined as the size of input data divided by the size of its compressed equivalent.) But dictionary coding is usually faster than statistical coding. This is especially important when choosing an algorithm for compressing data in real time. The compression algorithm must be able to generate compressed data at the speed supported by the underlying network layers divided by the compression ratio. Similarly, the decompression routine must be able to generate decompressed data at the speed of incoming compressed data divided by the compression ratio. If any of these two relations do not hold, the compression method will not increase the throughput of the communications link. However, in this case it might still be worthwhile to use compression. For instance, there might be a cost associated with the amount of data transmitted over the connection. Compressing the data would lower this cost. On a slow shared channel (e.g. a low speed packet radio channel) compression might be beneficial to the overall usage of the channel, even if the compression does not increase the individual throughput.

3. LZW compression

Ziv and Lempel [Ziv77, Ziv88] presented efficient compression methods that use dictionary coding. Several variants of the Ziv-Lempel methods exist, the most widely used seem to be the LZW method [Welch84] and its derivatives. A pleasing characteristic of these methods is that the encoder and decoder will build their respective dictionaries dynamically. The encoded data itself is used to build the dictionaries. At no point does the content of the dictionary have to be transferred.

An LZW encoder that operates on sequences of 8 bit bytes will emit codewords that are of a fixed width (at least 9 bits.) The dictionary is initialized to contain codewords representing single bytes with the values O-255. The operation of the encoder is to read bytes from the input stream, concatenating them into a string. It proceeds with reading bytes as long as it can find an entry for the string in its dictionary. If the string is not in the dictionary, it will be added, and a new codeword will be assigned to the new dictionary entry. Following this, the codeword for the previous version of the string will be emitted. The only difference between the previous and the current version of the string is the last byte. This last byte will now be used as the first and only byte in the string. The process is then repeated by reading yet another byte from the input stream.

The following example is in accordance with the original LZW specification, with the addition of a special ***flush*** axleword. Its use is described later. Let us consider the encoding of the string "ababc." The encoder will read the character 'a' from its input stream. The string is now "a", and since it is a one byte string, we know that it is always in the dictionary. Then 'b' is read. The string, "ab", is not found so it is added to the dictionary and the codeword for "a" is emitted. The string now becomes "b". The character 'a' is read. The string "ba" is not found, and is thus added to the dictionary. Then the codeword for "b" is emitted. The string is now "a" and 'b' is now read. The string **"ab"** is found in the dictionary, so yet another character, 'c' is read. "abc" is not in the dictionary, so it is added, and the codeword for **"ab"** is emitted, and the string becomes **"c".** At this point there are no more characters in the input stream. We are supposed to hold on to the string **"c"** at least until we have read one more character, but since the input stream is empty, we will simply transmit the codeword for **"c"** followed by a ***special flush*** codeword. We will refer to this from now on as the "flush" operation. The ***flush*** codeword is needed to indicate that the normal flow of compression has been broken by flushing the content of the string onto the output stream.

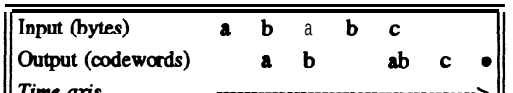

Figure 1.
This figure illustrates the synchronization between the encoder and the decoder. The decoder constantly lags one codeword behind the encoder. When the input stream ends, the encoder must emit the codeword of the string it was waiting to encode and emit the special *flush* codeword (denoted with a "•" here.) Otherwise the decoder will not receive the last codeword ("c" *in* this example.)

The decoder will build its dictionary only using the codewords it is receiving. When

receiving a codeword, it will take the first character in the string that the codeword represents, and concatenate it with the string represented by the previously received codeword. The resulting string will be added to the dictionary.

In the example above, the codeword for "a" is received. There was no previously received codeword, so no particular action is taken with respect to the dictionary. The codeword is now decoded into 'a' which is emitted to the decoded output stream. Then the codeword for "b" is received The concatenation of the "a" and the first character from the string "b" gives the string "ab". It is added to the dictionary and 'b' is sent to the output stream. Then the codeword for "ab" is received, causing the string "ba" to be added to the dictionary. Characters 'a' and 'b' are sent to the output. The codeword for "c" is received. The string "abc" is added to the dictionary, and 'c' is sent to the output. Now the special *flush* codeword is received. It causes the recording of the last received codeword to be cleared. Without it, should compression later resume, the decoder would construct a string beginning with "c" from the first codeword. That string would then incorrectly be added to the dictionary, creating havoc. Incidentally, since the decoder constructs new codewords using previously received codewords, the slightest data transmission error may cause large parts of the dictionary to become bogus.

It may sometimes happen that the decoder receives a codeword that it does not yet have in its dictionary. Consider for instance if the input stream consists of the string "ccccc". This will be encoded as the codewords for "c", "cc", and "cc" followed the flush codeword. When the first "cc" codeword is received, it will be unknown by the decoder. However, this situation can only arise if the encoded string is the same as given by the previous codeword followed by the first character in that same string. Evaluating this rule will yield the desired string, "cc".

4. Compression of transport protocol data

There has been recent studies [Jacobs90] on how to compress the headers of the TCP/IP protocol. This improves the efficiency of TCP/IP when sending very small packets, such as single characters typed at a keyboard. For larger size packets, the importance of TCP/IP header compression diminishes, as the importance of compression of the actual data that is sent by the transport protocol (e.g. TCP) increases.

Sometimes the transport protocol is only used to transfer a file. (Consider for instance the data TCP connection during an FTP file transfer.) In that case, almost any data compression method that can operate on the data "on the fly" will work. All variants of Ziv-Lempel compression will work in this situation. When the end of the file is reached, the encoder will perform the "flush' operation at least partially. It will encode the character string that it is currently holding and emit its codeword to the output. However, it is not necessary to emit any ***special flush*** codeword, since ***this was the*** last compressed data that was sent on the transport connection. The transport connection is sup posedly terminated shortly afterwards, so a special *flush* codeword would serve no purpose here.

Another situation that resembles a file transfer, is the transfer of electronic mail (an SMTP session for instance.) However, here there are interactive elements. In the case of STMP, for example, there is an initial dialogue between the sender and receiver of the message. When LZW or a similar method is used to compress the data, the decoder will always lag one codeword behind the encoder. This makes it necessary to perform the flush operation after each SMTP command and after each mail message. Consider what would otherwise happen. Most SMTP mail transfer agents will operate in stop-and-wait mode. They will send a command and they will not proceed until a response has been received Using standard LZW (non-flushing) compression, however, the SMTP server will not receive the complete command because its LZW decoder has not received the last codeword of the compressed command. The SMTP server will not do anything until it has received a complete command, but the mail transfer agent will not do anything either, because it believes that its latest command has been fully transmitted. A deadlock has occurred.

The same problem will arise when transmitting any kind of interactive data. An example application would be characters or lines of characters transmitted in transport protocol packets. (E.g. Telnet running in remote echo and local echo mode, respectively.) Even if there may not always be a deadlock, the lag of the LZW decoder is likely to cause some inconvenient effects. For instance, when using remote

echo, the echoed characters might not be received until several other characters have been typed.

A reasonable general solution to avoid the problem and possible deadlock described above, would be to perform the flush operation whenever the application program passes data for transmission to the layer below it, In order to keep software modules layered, it would be considered a good thing if the application program was as independent of compression as possible, and vice versa. Unless otherwise informed, the compression layer should assume that any data buffers are for immediate processing by the remote peer. Every buffer sent to the transport level would be "flushed." This would not be necessary when transferring files, or when doing other non-interactive operations, but the overhead would only be two codewords per data buffer. (The codeword of the last string, and the actual *flush* codeword) It would be possible to have the application program explicitly tell the encoder when to flush, but that could introduce complicated interactions between protocol layers.

Incidentally, suppose that Telnet is being run in remote echo mode. To echo each character after it has been typed would indeed be necessary to perform the flush operation for each character. Since the flush operation always generates two codewords, this would cause the output stream to be between 125% and 300% larger (for 9 to 16 bit wide codewords) than the input stream. This negative compression is of course not caused by something inherent in the actual LZW algorithm, but merely by the need to immediately flush any data. There are however situations when LZW will generate negative compression, independently of any flushing.'

5. An experimental implementation

The author implemented a data compression layer on top of TCP for the KA9Q Internet Package [Karn87] for MS/DOS. The implementation is based upon the original LZW specification but with the addition of the *flush* codeword for the flush operation. Another difference is that variable size codewords are used. The application program is able to specify the maximum size of the codewords, which must be in the range of 9 and 16. The encoder begins with 9 bit codewords and increases the size as needed. When the limit is reached, a *special clear* codeword is emitted. Both the encoder and decoder will then clear their dictionaries and revert to 9 bit wide codewords. These mechanisms for handling variable size codewords, and for handling the exhaustion of the dictionary are very similar to the mechanisms used by the LZW variant in the GIF graphics format [GIF87].

A reason why one would like to limit the maximum size of the codewords is to avoid possible memory limitations at either peer. The KA9Q program, for instance, does not leave much spare memory. The experimental implementation provides a tradeoff between memory usage and speed. When minimizing the memory usage, each dictionary entry uses three bytes. This is accomplished by observing that each codeword can be expressed as the string of another codeword followed by a single character. Thus, out of the three bytes, two are used as a pointer to a previous codeword, and the third represents the character that should be appended to the string of the previous codeword.

Still, when using only three bytes per entry, 16 bit codewords would require a dictionary size of 192 kbyte. If compression is used in both directions, the memory requirements would be doubled. To prevent accidental memory exhaustion, the encoder and decoder exchange their recommended maximum codeword size values. The smallest of the two values is used by both parties.

The decoder can construct its dictionary so that it can quickly look up a string by its codeword. If each entry only consists of one character and a pointer to another codeword, the decoder will get the string in reverse order. But the decoding will still be fast. The encoder, on the other hand, must search its dictionary to see if a given string is already in the dictionary. This is a very time consuming process when the dictionary grows in size. A way of avoiding this problem would be to restrict the size of the codewords to a size that generates reasonably small dictionaries, such as 12 bits. It has also been argued that using codewords larger than 12 bits has minimal impact on the efficiency of the

[1] An example of this is when compressing a file that has already been compressed. To avoid this problem it would be necessary for the application to detect that its data has already been compressed and to disable any further compression by lower layers.

compression. English ASCII text will usually give a compression ratio of 2:1 already at small codeword sizes.

The dictionary can be constructed in different ways to make searching for a string faster. The experimental implementation provides a "fast" mode where a hashing mechanism is used to speed up searching. Dictionary entries do now have to be kept as a linked list. This causes each entry to be 5 bytes, a relatively significant increase in memory usage.

The actual performance of the KA9Q LZW implementation depends mainly on the type of hardware. the program is being run on. Using an 8 MHz AT clone there were no problems with keeping a 1200 baud SLIP link saturated when compressing with up to 12 bit wide codewords using either "fast" or "compact" modes of operation.

6. The V.42bis compression standard

The CCITT has published a recommended data compression standard called V.42bis [CCITT90]. It is being implemented as an integral part of many telephone modems. The compression method is in essence LZW, and one is assumed to run it on top of an error free communications link. This is usually achieved on telephone modems by using the V.42 error correction protocol. But V.42bis may also be run on top of some other protocol that provides an error free link, such as TCP.

V.42bis has many similarities with the LZW implementation in the KA9Q package described above. It also uses variable size codewords and a special *flush* codeword for the flush operation.

In V.42bis, the *flush* codeword is always added to the output buffer that is being flushed' and then the buffer is padded into an integer number of bytes. This is not entirely necessary, however. The LZW implementation for the KA9Q package uses the *flush* codeword to pad the output buffer into an integer number of bytes. This means that the part of the flush codeword will be in one output buffer and the remainder will be at the beginning of the next output buffer. This works because the *flush* codeword does not need to be processed by the decoder until there are some more codewords to decode. By padding with "blank" bits, V.42bis wastes up to '7 bits (or 3.5 bits on average) per output buffer when compared to the KA9Q implementation.

CCITT specifications follow the tradition of trying only to specify the operation of the encoder. The decoder should just perform the logical inverse of the encoder. In trying to keep up with this tradition, the CCITT had to modify the way the decoder is assumed to handle the receipt of codewords that it does not yet know about. As described previously, if the input stream consists of the string "ccccc", a standard LZW encoder will encode it as the codewords for "c", "cc", and "cc" followed by the flush codeword. The first "cc" will not be in the dictionary of the decoder, but it knows that this can only happen for repeated character strings. The decoder can find out which string the unknown codeword must represent by using a relation between consecutive codewords that holds true in this case. V.42bis, however, differs from the original LZW specification in that the encoder will refrain from emitting a codeword that is identical to the latest codeword it added to its dictionary. (See p. 27 in the V.42bis specification for the exact algorithm.) In this example, the V.42bis encoder will emit the codewords for "c", "c", "cc" and "c" followed by the *flush* codeword. This is one codeword more than what would be needed by systems following the original LZW specification (such as the KA9Q implementation.) In many cases, however, both V.42bis and original LZW produce the same amount of codewords for a given string of repeated characters.

There is a provision in V.42bis for the application program to temporarily disable compression. This is very useful when the flush operation would otherwise have to be performed after every single character. (E.g., when transmitting keyboard input in remote echo or full duplex mode.) Without this facility, those cases could lead to a quadrupling of the bit stream, as mentioned previously.

The V.42bis specification includes the data structure, called trie [Knuth73], to be used in the dictionary by the encoder. The trie allows for efficient dictionary lookups by using pointers to "child", "parent" and "sibling" codewords. This causes each dictionary entry to use at least 7 bytes. When the dictionary is full, it is not cleared as in GIF or in the KA9Q implementation[2], but rather pruned in a circular fashion. The pruning algorithm begins with the oldest codewords, and deletes those that have no children or siblings. The result is very similar to a

Least Recently Used algorithm.

7. Alternative algorithms

LZW derived algorithms are very popular, at least until recently. The LZW algorithm has however recently been patented by Unisys. LZW implementations also risk infringement of the Miller and Wegman patent which is held by IBM. (Miller and Wegman discovered LZW independently from Welch [Miller84].) As far as the author knows, neither of these patents have yet been tried in court

There are several other dictionary algorithms based on the original Ziv-Lempel methods. Although they may not be as fast or efficient as LZW, implementations of these algorithms do not yet risk patent infringement. There are also several promising statistical compression methods, in particular PPMC [Moffat88]. Although efficient, statistical compression methods often require lots of processing. PPMC seems to be a good compromise and it has been shown to perform well on medium range workstations. [Cate91]

8. Conclusion

Data compression is a viable method for increasing the throughput of a communications link. Care must be taken when choosing a suitable implementation of an algorithm. The implementation may otherwise only decrease the amount of data transferred while imposing delays that lower the throughput. Special consideration must be taken when choosing an algorithm for compressing interactive traffic. An LZW implementation for interactive TCP data in the KA9Q Internet package has been described. V.42bis solves some of the drawbacks of the KA9Q implementation, but has also been shown to use less efficient coding than KA9Q.

9. References

Bell89 Bell, T., et al. "Modeling for text compression." *ACM Computing Surveys,* Vol. 21, No. 4, December 1989.

Cate91 Cate, V., Gross, T. "Combining the Concepts of Compression and Caching for a Two-Level Filesystem." School of Computer Science, Carnegie Mellon University, 1991.

CCITT90 "CCITT Recommendation Data Compression Procedures for Data Circuit Terminating Equipment (DCE) using Error Correction Procedures." *Vol. WI, Rec. V.42bis,* Geneva 1990.

GIF87 "GIF. Graphics Interchange Format (tm.)" CompuServe Inc, June 1987.

Jacobs88 Jacobson, V. "Congestion Avoidance and Control." *Proceedings of Sigcomm* '88. ACM, August 1988.

Jacobs90 Jacobson, V. "Compressing TCP/IP Headers for Low-Speed Serial Links." *RFC-1144,* February 1990.

Karn87 Karn, P. "The KA9Q Internet (TCP/IP) Package: A Progress Report" *6th Computer Networking Conference.* ARRL, August 1987.

Knuth73 Knuth, DE. "The Art of Computer Programming." Vol. 2, "Sorting and Searching." Addison Wesley, Reading, MA, 1973.

Miller84 Miller, V.S., Wegman, M.N. "Variations on a theme by Ziv and Lempel." In "Combinatorial Algorithms on Words." (Apostolico, A., Galil, Z., *editors) NATO ASZ series,* Vol. F12. Springer-Verlag, Berlin 1984.

Moffat88 Moffat, A. "A Note on the PPM Data Compression Scheme." *Tech. Report 88/7,* Dept. of Computer Science, University of Melbourne, July 1988.

Storer88 Storer, J.A. "Data Compression: Methods and Theory." Computer Science Press, Rockville, MD, 1988.

Welch84 Welch, T. A "Technique for High-Performance Data Compression." *IEEE Computer,* Vol. 17, NO. 6, June 1984.

Ziv77 Ziv, J., Lempel, A. "A Universal Algorithm for Sequential Data Compression." *IEEE Trans. Inf.* Theory, Vol. IT-23, No. 3, May 1977.

Ziv78 Ziv, J., Lempel, A. "Compression of Individual Sequences via Variable-Rate Coding." *IEEE Trans. Inf. Theory,* Vol. IT-24, No. 5, September 1978.

[2] V.42bis does provide a *clear* codeword, however. But its use is left undefined.