Robust Data Compression of Network Packets

Sean Dorward and Sean Quinlan
Bell Labs, Lucent Technologies

Abstract

This paper describes an approach for compressing data packets that enables inter-packet compression without the drawback of multiplying the effect of packet loss. By adding an acknowledgment scheme, the sender can limit the history state used by the compression algorithm to those packets that have been correctly received. A vector identifying the packets used as history is included in the compressed packet, enabling the receiver to reconstruct the history state required to decompress the packet. This approach improves the compression achieved compared to stateless schemes, while retaining robustness in the face of packet loss or reordering.

The paper reports several simulation experiments using our approach. We first describe an implementation based on the publicly available zlib implementation of the popular deflate compression format. We then describe the implementation of a Lempel-Ziv '77 variant called *thwack* that is more efficient at handling the unpredictable history state used to compress and decompress packets.

Introduction

We are interested in improving the performance of packet networks. When available computational resources are large compared to the network bandwidth, it can be beneficial to compress data packets before transmitting them. We must consider both the speed of a compression algorithm and its ability to compress. Too slow an algorithm may reduce performance, while insufficient compression limits any potential gain.

As an additional complication, many networks are unreliable -- they drop or reorder packets. The Internet is an example of such a network. If compression introduces dependencies between packets, it may be impossible for the receiver to decompress a packet when a previous packet is lost. Existing solutions to this problem include:

1) Make the network reliable. Even unreliable networks typically provide a reliable end-to-end transport service; on the Internet TCP is such a service. Compressing packets at the transport level is feasible, but must be applied end-to-end, and often requires cooperation by the application. Also, this approach is not appropriate for transparent compression over a link in the middle of the network, a case in which we are interested.

2) *Stateless compression*: Compress each packet independently. Since each packet is independent, it can be always be decompressed by the receiver. Unfortunately this independence fundamentally interferes with the effectiveness of compression. Compression is achieved by finding the commonality of between various parts of the data and exploiting this commonality to send less information. Stateless compression only examines the data in a single packet, which reduces the compression ratio. In particular,

this approach cannot remove the large amount of reducnacy found in the network headers of adjacent packets. Ipcomp [1] is an example of stateless compression.

3) *Streaming compression*: Assume reliable delivery and use a reset mechanism when this assumption is violated. When a packet is lost, the receiver discards each subsequent packet until compression is reset. After the reset, future packets will not depend on prior packets and decompression can resume. PPP's Compression Control Protocol [2], and the IP Header Compression protocol for UDP packets [3] use variants of this approach. Streaming multiplies the effect of packet loss: one packet lost in the network causes the receiver to lose many packets. For networks such as the Internet, where reliability can be quite low, this multiplying of the loss rate makes streaming compression unattractive. In addition, the Internet often reorders packets. Since it is difficult to distinguish packet reordering from packet loss, reorders often cause multiple packet losses when using streaming compression.

In this paper we introduce *acknowledged compression*, a novel approach combining the robustness of the stateless scheme with the greater compression ratio of the streaming scheme. By adding an acknowledgment mechanism, the sender can determine which packets were received correctly, and thus exploit inter-packet dependencies without multiplying the effect of a lost packet.

We describe an implementation of acknowledged compression using the publicly available zlib package; report its limitations; and develop a new algorithm, *thwack*, with superior performance.

History State

In the streaming scheme, the compression algorithm typically generates compressed packets that are dependent on some set of previously sent packets. We shall refer to this set of previous packets as the *history state*. For algorithms such as Lempel-Ziv '77 [4], this history state is directly used as the dictionary for compression. For algorithms like Lempel-Ziv '78 [5] and PPM [6], the history state is transformed into some internal state which is then used as the basis of compression.

To limit the memory requirements needed for the compressor and decompressor, the amount of history state is typically bounded. This can be implemented as a sliding window, or by simply truncating the history when the limit is reached. For example, the deflate [7] compression format allows a maximum of 32 kilobytes of history with a sliding window scheme. Restricting the history size can have a significant detrimental effect on the compression ratio, but for larger history sizes this effect becomes negligible.

The main difference between stateless and streaming compression of network packets is the history state they use: stateless compression allows no history state, while the streaming compression allows any previous packet as history. From this perspective, the stateless compression scheme appears overly pessimistic; we expect at least some packets to have reached the receiver, and by allowing no history, the effectiveness of compression is decreased. In contrast, the streaming scheme is overly optimistic and

assumes no packets will be lost; high compression ratios can be achieved at the cost of a potentially large increase in the packet loss rate.

We propose an intermediate solution. Suppose the sender has some information about which packets have been successfully received. Assuming both ends have retained these packets, the compression algorithm can use these packets as history state with the expectation that the receiver will be able to decompress the packets. The decompressor needs to know which packets were used as history during compression, but this information can be included with the transmitted packet. By using this selected history state, we get good compression while avoiding the problems caused by packet loss that are inherent to streaming compression.

An acknowledgment scheme provides the sender with information about which packets have been received. The acknowledgments can be piggybacked on data packets in the opposite direction, or sent in dedicated packets. Note these acknowledgments are only used for the purpose of determining a history state for compression and not for re-transmitting lost packets; they are in addition to acknowledgments used by higher level transport protocols. Moreover, if acknowledgments are delayed or lost, the compressor will have less information about the state of the decompressor, but it will still be able to generate decompressible packets.

Our approach requires a compression algorithm that can compress and decompress a packet given a specified history state. Although this requirement is a little unusual, most algorithms can be adapted via the following strategy. For each packet, reset the compressor to its initial state and then bring the compressor to a state that corresponds to compressing the sequence of history packets. The current packet can then be compressed and sent across the network. When the compressed packet is received, the decompressor examines the packet to determine which history packets were used. The state of the decompression algorithm is reset and then brought forward using the indicated history state. Finally, the current packet is decompressed. Later in this paper we describe a Lempel-Ziv ’77 (LZ-77) style algorithm that implements this strategy efficiently.

Due to space constraints, a receiver must eventually discard history packets. Given this limitation, the receiver may not be able to decompress a packet even though it was compressed with a valid history state. This problem arises when a packet is reordered and delayed relative to a large number of subsequent packets, causing the receiver to discard a required history packet. To avoid this problem, the amount of the history state retained by the receiver should be larger than that of the sender by a quantity proportional to maximum expected propagation delay of the network.

Packet Header

To implement our compression scheme we prefix each data packet with a header that includes a sequence number, a history vector and an acknowledgment vector. This header is overhead compared to the stateless scheme. The streaming scheme requires just the sequence number so it can detect lost or reordered packets.

The sequence number is used by both the sender and receiver to identify packets. The number of bits used for the sequence number should be sufficient to avoid wrapping during the maximum round trip time of the network. Our implementation uses 24 bits.

The history vector describes the set of previous packets used for compression. Since only acknowledged packets are used for history, we expect that there will be a delay related to the round trip time of the network between the time a packet is sent and the time that it will be used as history. Therefore we encode the history vector using an offset and a bit mask; the offset is subtracted from the packet's sequence number to give the sequence number of the most recent history packet; the bit mask identifies additional history packets directly preceding this packet. Our implementation uses 8 bits of offset and 8 bits of mask, limiting the history state to at most 9 consecutive packets within the last 264.

The acknowledgment vector describes a set of recently received packets for the opposite direction packet stream. It contains the sequence number of the most recently received packet and a bit mask describing the status of the directly preceding packets. We use 24 bits for the sequence number and 8 bits for the mask. Note, with this choice, a single acknowledgment provides the same amount of information as the history vector and thus can completely update the sender's history state. Since the acknowledgment vector is not used by the decompressor, it can be included in the compressed section of the packet.

Experiments based on Deflate

This section provides some experimental results of simulations of our compression scheme using deflate as the underlying compression algorithm. Deflate is a popular compression format that is the basis of many file formats including gzip, zip, and png. In the context of network protocols, it has been used with PPP [8] and Ipcomp [9]. Deflate combines LZ-77 and Huffman encodings. The following experiments where done using the publicly available zlib implementation of deflate with the compression level set to the default value.

We using an artificial sequence of data packets to compared the compression schemes. We created a byte sequence by concatenating the 14 commonly used files of the Calgary corpus in alphabetical order, and then divided the sequence into fixed sized packets. Real packets are obviously quite different than our artificial packets: they include headers and vary in size. However, the results obtained with the artificial packets are easily reproduced and do not different qualitatively from those we have obtained when using our compression scheme on an actual network link.

As described above, the acknowledgments are piggybacked on packets flowing in the opposite direction. For our particular applications, there is sufficient traffic in each direction to make this piggybacking effective; other applications may require dedicated acknowledgment packets. Our simulations model the acknowledgments as arriving at the compressor after a delay of some fixed number of packets. This would correspond to a scenario of a constant rate stream of packets in both directions with a fixed latency in the

network. Obviously reality is a lot more variable than this model, but we believe it provides useful insight into the performance of our approach.

Figure 1. compares the effect of delay on the number of output bits per input byte for packet sizes of 1600 bytes and 125 bytes. The bits per byte of the stateless and streaming schemes for both packet sizes are also indicated. The interesting feature of these graphs is the effect the acknowledgment delay has on the compression ratio. As the delay increases, the compressor must use history packets that are further away from the current packet and the graphs show this has a significant effect. Even with delay, however, our approach gives substantial improvement over the stateless approach while retaining robustness in the face of packet loss.

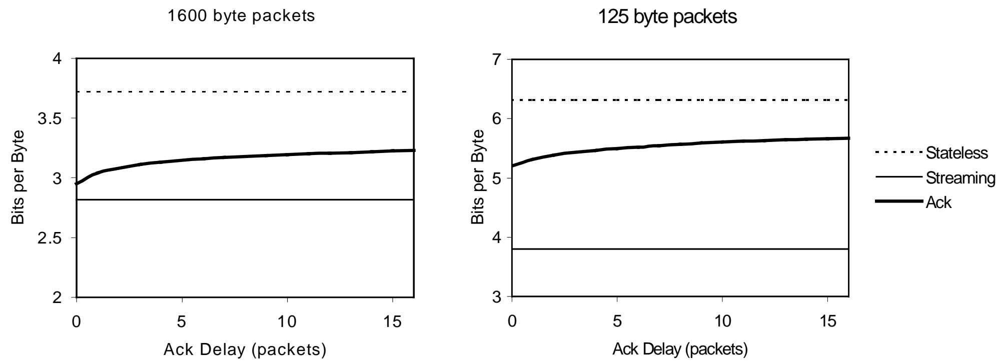

Figure 1. Effect of delay on compression ratio.

Not surprisingly, the streaming approach gives the best compression. However, it does not take into account the effect of packet loss. The streaming approach will multiply the packet loss rate by the delay, since this is the time required to send a reset to the compressor. This multiplication of loss rate has a dramatic effect for higher level protocols, such as TCP, that assume packet loss corresponds to network congestion. In our experience this combination can even render a network unusable.

One might expect the zero delay acknowledgment and streaming schemes to give identical compression ratios. However, our encoding of the history vector limits the history state to 9 previous packets, significantly less than the 32 kilobytes of history used in the streaming version. This gives less compression, especially for small packets. A more expressive encoding of the history vector would enable using larger history states. Note, however, that the size of the history state also effects the speed our implmenations.

Compression Speed

The major motivation for using compression on a network is to reduce the data transmission time. Consider a model of the compressed communication link in which we assume that each packet has fixed size of n bits that compresses to n' bits. Further, suppose compression takes t_c seconds per packet and decompression takes t_d seconds. If the communication link has a bandwidth of B bits per second, the time to send a single uncompressed packet over the link is n/B and the time to send a compressed packet is

$t_c + n'/B + t_d$. The *bandwidth ratio* r of the compressed to uncompressed link is computed by dividing these times, giving

$$r = \frac{n}{B(t_c + t_d) + n'} .$$

This ratio is a conservative measure of the increase in bandwidth. If processing is overlapped with transmission, and t_c and t_d are both less than the time to transmit the compressed packet, then the improvement in the effective bandwidth is n/n'. One the other hand, this bandwidth is only achieved over a large sequence of packets. Moreover, if r is less than one, the latency for a particular packet will have increased. To avoid these hidden costs, we used the more conservative ratio r in the following experiments.

The ratio of communication bandwidth to available computational resources is the decisive factor in deciding if compression should be used on a link and how a compression algorithm should be configured. To give a representative sample of likely configurations we performed our experiments on a Pentium II 400Mhz running Windows NT 4.0 with simulated communications bandwidths of 10kbs, 100kbs, 1Mbs, and 10Mbs. Naturally, varying the device used to perform the compression requires commensurate scaling of these bandwidths.

Figure 2 charts the bandwidth ratio for 1600 and 125 byte packets using stateless, streaming, and our acknowledgment approach with a delay of 8 packets. We used the compression ratios from our previous experiment, and measured the actual compression and decompress times for each scheme. Using these results, we computed the bandwidth ratio for the simulated communication links using the above equation.

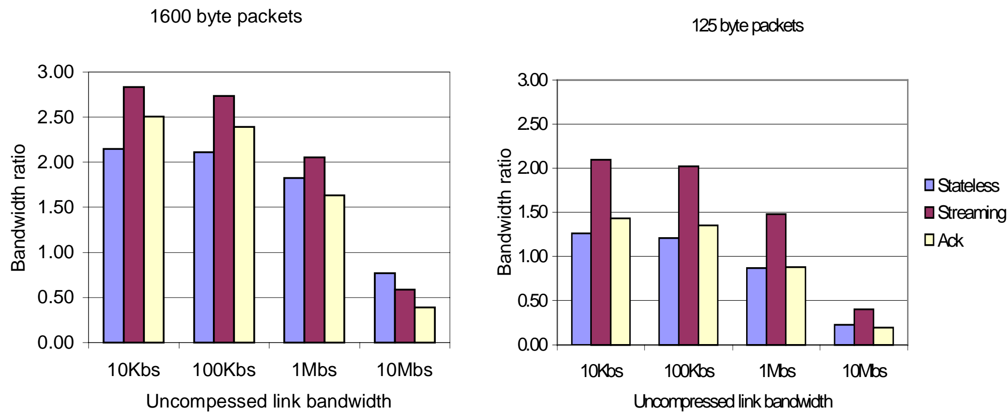

Figure 2. Bandwidth ratio for zlib.

For small link bandwidths, the time to compress and decompress packets is negligible compared to the transmission time. In this situation, the bandwidth ratio approaches the compression ratio. Since the acknowledgment approach gets better compression than the stateless scheme, we achieve a higher effective bandwidth.

As the link bandwidth increases, processing time becomes significant and the bandwidth ratio goes down. Zlib takes more time to compress a packet related with a larger history state; since the stateless scheme has no history state, it compresses packets in less time. For 10Mbs this advantage is sufficient to give the stateless scheme the best bandwidth ratio for larger packets. Note for 10Mbs all the approaches have a bandwidth ratio less than one; thus it is questionable whether compression is appropriate.

As can be seen, compression is not nearly as effective for smaller packets, especially at higher bandwidths. Part of the problem is that deflate is a poor format for small packets. In particular, each packet is compressed with either static or dynamic Huffman encoding. The static encoding appears to be poorly designed, so the dynamic encoding is often used. While Huffman compression is worthwhile for offline compression of large files, the overhead of computing and transmitting the Huffman dictionary in each packet is too large. To make matter worse, the deflate format requires a large overhead of more than 8 bytes to terminate a packet.

The interface provided by zlib allows the specification of a compression level. This parameter determines how much work is performed when finding a potential match in the history, enabling a tradeoff between compression speed and compression ratio. For high link bandwidths it is typically advantageous to set this level to its fastest setting rather than the default setting we used above; the loss in compression ratio is more than compensated by the reduction in compression time. Unfortunately, for our acknowledgment scheme it turns out the compression level has little effect on the bandwidth ratio. To implement our approach using the zlib library, we use the deflateSetDictionary function to initialize the history state for each packet. This function takes the history and builds a hash table. Since the history is 9 times larger that the packet we compress, the time to build the hash table dominates the time to compress the packet, thus reducing the speedup from using lower compression levels.

Thwack

To overcome the problems of using deflate, we designed and implemented a new LZ-77 compressor call *thwack*. Unlike deflate, thwack uses a well chosen static encoding thus avoiding the cost of computing and sending Huffman tables. The implementation also deals efficiently with the variable history state of our acknowledgment scheme. These changes provide much improved bandwidth ratios for high speed links and small packets.

We use a simple variable-length coding for match lengths, offsets, and literals. The encoding was designed by analyzing a large sample of real data. It performs well on our artificial data, but was not trained specifically for it.

The details of the encoding are available [10]. All phrases start with a match length; literals are encoded with a zero length match followed by an encoding of the literal. The smallest non-zero length match is three bytes; the maximum match length is limited by the packet size, not the encoding scheme.

Literals are encoded using a simple predictive scheme. Long strings of printable ASCII characters are encoded using 7 bits; other literals are typically encoded using 8 bits. Match offsets are limited to 14 bits, and encoded in two pieces: a range, and the position within that range. The range determines the number of bits used to encode the position, with fewer bits used to encode closer offsets.

Thwack History Implementation

The core of an LZ-77 implementation is the method used to find matches. A match can be found in the part of the current packet that has been compressed or any of the packets that makes up the history state. The zlib implementation uses a single hash table to speed the search for a match. This design is suitable for the streaming case, but when used with our acknowledgment scheme requires that the hash table be rebuilt for each packet.

Our implementation of thwack maintains a separate hash table for each packet, which is constructed as the packet is compressed. The resulting table indexes all three byte strings in the packet; it does not index data in any other packet. Since each packet's table depends only on the data in that packet, an arbitrary set of tables is a valid index for the corresponding packets. To search for a string, we query the table for current packet, followed by the tables for each of the packets in the history state, starting with the most recently sent packet. To limit to cost of examining multiple hash tables, the number of packets in the history state must be reasonably small. For small packets, this means only a small amount of history data is used. Our method, however, avoids the expensive hash table rebuild required by the zlib implementation.

Our hash tables are otherwise similar to zlib's. We maintain singly linked offset chains terminated by an invalid entry, and recycle hash tables by changing the valid offset range for the table, not by clearing the entire table.

Like zlib, we provide a compression level parameter to determine the maximum number of strings that are checked when looking for a match, allowing adjustment of the speed versus compression tradeoff.

Using thwack to compress the individual files in the Calgary corpus, we achieve an average compression of between 3.06 and 3.48 bits per byte, depending upon the compression level. Substituting zlib for thwack, and maintain the same history state and packet size limitations, we average between 3.03 and 3.33 bits per byte for comparable compression levels.

Figure 3 compares zlib and thwack using our acknowledgement scheme with a delay of 8 packets. For each compression algorithm, results for two setting of the compression level are given. Recall, to generate these results, the compression ratio and compression time are measured and from this we compute the bandwidth ratio for different links. As can be seen, thwack almost always outperforms zlib for comparable compression levels. For small packets, thwack is clearly superior.

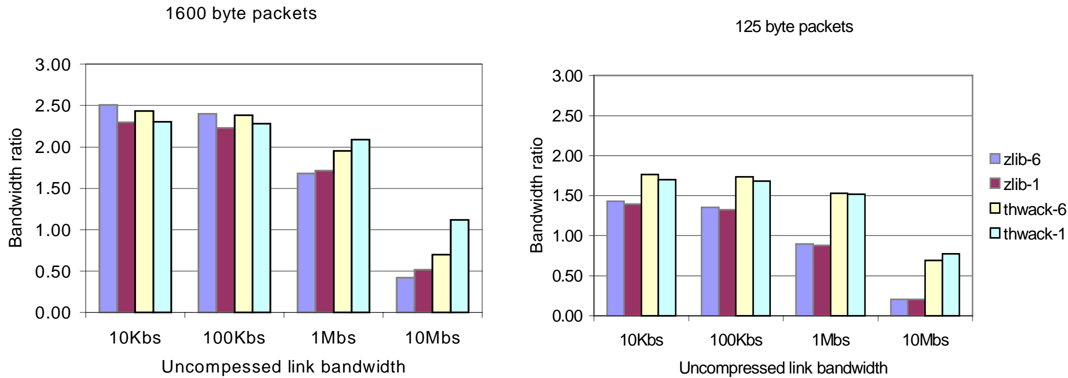

Figure 3. Bandwidth ratio for zlib and thwack

Conclusion

The acknowledgment scheme proposed in this paper enables the compression of network packets with inter-packet dependencies without multiplying the loss rate. We believe this scheme provides a good compromise between the simplicity and robustness of stateless compression and the high compression ratio of streaming compression.

For compression of network packets, speed is important especially if the link bandwidth is large relative to the available computational resources. We have described an LZ-77 variant, called thwack, that can efficiently compress small packets and handle the unconventional history state used in our scheme.

One interesting attribute of the zlib implementation of deflate and our implementation of thwack is that they provide a compression level parameter, enabling a tradeoff between speed and compression ratio. In the context of compressing network packets, the bandwidth of the uncompressed link should really determine the setting of this parameter, since it directly influences the effective bandwidth that is achieved. It would appear interesting to investigate dynamically setting the compression level based on feedback from the system about the actual link bandwidth.

We have not experimented with intelligent selection of the history state. In a network with multiple connections, packets will probably compress better with a history state composed of other packets from the same connection.

The advantage of the acknowledgment scheme is that it can tolerate lost or reordered packets. However, the delay in receiving acknowledgments and a reduction in the history state size result in a lower compression ratio than the streaming approach. Given the acknowledgments from the receiver, it is possible to sender to monitor the reliability of a link. One possible extension of our scheme is to have the sender to notice when the link reliability reaches some threshold and switch to the streaming approach, improving the compression ratio. If the link subsequently becomes less reliable, we could switch back to the acknowledgment scheme.

References

[1] A. Shacham, R. Monsour, R. Pereira, M. Thomas, “IP Payload Compression Protocol (IPComp),” RFC 2393, December 1998.

[2] D. Rand, “The PPP Compression Control Protocol (CCP),” RFC 1962, June 1996.

[3] M. Degermark, B. Nordgren, S. Pink, “IP Header Compression,” RFC 2507, February 1999.

[4] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inform. Theory*, vol. IT-23, pp. 337-343, May 1977.

[5] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530-536, September 1978.

[6] A. Moffat, “Implementing the PPM data compression scheme,” *IEEE Trans. Communications*, vol. COM-38, pp. 1917-1920, November 1990.

[7] P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3,” RFC 1951, May 1996.

[8] J. Woods, “PPP Deflate Protocol,” RFC 1979, August 1996.

[9] R. Pereira, “IP Payload Compression Using DEFLATE,” RFC 2394, December 1998.

[10] S. Dorward and S. Quinlan, “The Thwack Compression Format,” In preparation.