Adaptive Header Compression for Wireless Networks

Changli Jiao
Department of Electrical and Computer Engineering
Wayne State University
Detroit, MI 48202
Email: changli@katha.eng.wayne.edu

Loren Schwiebert
Department of Computer Science
Wayne State University
Detroit, MI 48202
Email: loren@cs.wayne.edu

Golden Richard
Department of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: golden@cs.uno.edu

Abstract

TCP/IP header compression has long been used to send information efficiently and to improve the response time of communication systems. The wide deployment of the new Internet Protocol, IPv6, puts more demands on header compression, since IPv6 normally yields packets with much larger headers than IPv4 does. At the same time, header compression is necessary on wireless links, which are always considered relatively limited resources. However, packets sent over wireless channels are prone to be corrupted. This usually deteriorates the performance of header compression. In addition, the recently noticed high frequency of some computer networking problems makes the performance of header compression even worse. These problems include packet reordering and packet errors that avoid link layer error detection. In this paper, we analyze the influence of these problems on existing header compression algorithms. We also propose a new algorithm, which is adaptive to the wireless channel as well as the packet size. This new algorithm will thus achieve a better tradeoff between compression ratio and error propagation, which will give better overall performance. Even though we focus on wireless links in this paper, the adaptive algorithm would also be suitable for wired links where header compression performance is influenced by packet reordering and errors passing link layer detection.

1 Introduction

Header compression has long been an important issue in the TCP/IP protocol suite. After compression, most packets are expected to contain shorter headers to be transferred over the communication medium. In this way, compressed packets can usually reach the destination after a shorter period of time compared to communication without compression. From the standpoint of the communication medium, the channel usage will be improved, which means more packets can be carried during a given period of time.

TCP/IP packets with compressed headers normally traverse only a single link. Because the router needs the whole header to perform routing, making the router understand the compressed header would involve too much change or may not even be feasible. So there is a compressor on one side of the communication channel and on the other side, a decompressor. However, the compression and decompression process could introduce errors. Usually, headers are compressed based on some information known by the decompressor. Correctly decompressed packets refresh the information on the decompressor side, which allows the following packets to be recovered. A corrupted packet obviously can not be decompressed and has to be dropped. And this packet's failure to be decompressed could cause the decompressor to become desynchronized. Thus, subsequent packets received by the decompressor may also be dropped even though they are transmitted correctly. This is called *error propagation*, which means the packet dropping is caused by errors in previous packets instead of this packet itself. Continuously discarding packets on the forward path, due to error propagation, results in ACKs not being sent. Similarly, error propagation on the reverse path will prevent the TCP sender from receiving the ACKs. In the TCP sender, ACKs are used to indicate network conditions. The sender will even transmit packets at a lower speed if the gap between ACKs is too big. This will have two side results. First, the efficiency of compression is lower, or perhaps even more bits need to be sent compared to transmitting without compression. Second, the correct information is received later, maybe even later than transmission without compression. This could be harmful to real time or delay-sensitive applications. In some real time situations, applications can still use the damaged information, in which case the tradeoff between delay and having to use erroneous information needs full consideration. These are the bad influences when the decompressor has to drop packets. If, for some reason, the decompressor can not detect that a packet was corrupted, worse results emerge. Wrong information will be passed to higher layers. Even though there may exist other error detection mechanisms on higher protocol layers, the correctness should be questioned, since it is applied on an error base.

There are a number of proposals for TCP/IP header compression [7] [3] [5] [6]. At the same time, recent research shows that packet reordering is a common phenomenon in modern computer networks [1], and that there are various sources of errors that are passed to the transport layer [11]. In this paper, we will analyze the performance of header compression proposals under these two phenomenon. We will also present a new algorithm for header compression over wireless networks, which achieves a good tradeoff between throughput and compression ratio.

2 Previous Work

In the early stage of computer networking, most of the popular mediums for connecting home PCs to the Internet were low speed serial links. One example is the modem, which could support 300 to 19,200 bps. At that time, people wanted to conserve the bandwidth of this kind of link, and be able to connect with the Internet faster. So, in 1990, Van Jacobson proposed a TCP/IP header compression algorithm for low-speed links [7]. This algorithm was designed for TCP running over IPv4. Jacobson carefully analyzed how the TCP/IP headers change for each packet throughout a connection. By using the changing pattern, the algorithm can compress the usual header size of 40 bytes down to 4-17 bytes. VJ compression is a proposed standard in IETF, and it has been deployed very widely.

Even today, the bandwidth provided by modems is still quite limited, less than 100Kbps. At the same time, the communication mediums used in computer networks have changed much. Wireless is becoming a popular way for connecting computers to the Internet. The bandwidth of wireless links varies over a wide range. Wide-area networks, due to constraints of the physical medium, can only provide low-speed bandwidth, on the order of 10Kbps. Some emerging new protocols will support higher data transmission over conventional voice channels [10], on the order of 100Kbps or several Mbps. The other kind of network, local-area wireless networks, operates on a smaller geographical area, which in turn faces a better physical environment. It can operate at a few Mbps, which is also expected to be improved. But, due to the regulatory limitations on the use of radio frequencies and inherent information carrying limits, all wireless links will remain scarce resources even after these new protocols are fully deployed. Again, people need to conserve the bandwidth and to use the link more effectively. Besides the change of network mediums, Internet protocols also face changes. The first one is that the Internet Protocol, IPv4, will finally be replaced by a new protocol, IPv6 [4]. In IPv6, the address size will be increased from 4 bytes to 16 bytes. In addition, some fields of IPv4 are removed from the basic IPv6 header in order to speed the packet processing on routers. So the basic IPv6 header is 40 bytes, while the minimum IPv4 header is 20 bytes. Moreover, various extension headers can be added to the basic IPv6 header to provide various routing, security, and other features. The larger header will definitely add more challenges on wireless links. The second one is particular to mobile users. The need for wireless data communication arises partially because of the need of mobile computing and partially for some specialized applications. Under some cases for mobile computing, it is required that one IP header be encapsulated in another one, or a routing header is added to an IPv6 header. Even though mobility provides convenient access to the Internet and has become an important use of wireless networks, fulfilling this task requires more bandwidth, which puts more pressure on wireless networking resources.

In 1996, Degermark et al. proposed a header compression algorithm for UDP and TCP for IPv6 networks [2]. The header compression for TCP is quite similar to VJ compression. The basic idea of compression is not transferring redundant information whenever possible. Thus, the number of bits in a compressed header will become less. The TCP/IP header fields can be divided into the following groups:

- Constant fields. These fields usually do not change during the lifetime of a connection, for example, source address, source port, destination address, and destination port.
- Inferable fields. These fields can be inferred from other fields, like the size of the frame.
- Delta fields. These fields are expected to change only slightly from the fields of the previous packet.
- Random fields. These fields have no relationship with other packets, like the checksum in TCP header.

Constant fields can be eliminated from most packets. Instead, a unique number, the Compression Identifier (CID), is assigned and added to the packet header to identify these fields. Inferable fields will not be transferred at all. The decompressor will calculate and fill them in. Delta fields will be transferred using only the differences, which can be expressed with fewer bits and are referred to as "delta" values. No change will be made to random fields, since there is no way to predict or calculate them. Thus, several new kinds of packet types are defined under header compression. The packet types defined in [7] [3] [5] [6] are quite similar. The compression method uses the following packet types in addition to the IPv4 and IPv6 packet types:

- UNCOMPRESSED_TCP - the packet with a full TCP/IP header as well as the CID.
- COMPRESSED_TCP - the packet with all fields compressed and the CID added.
- SEMICOMPRESSED_TCP - the packet with all fields compressed except delta fields and with the CID added.

The decompressor records or revises fields when receiving UNCOMPRESSED_TCP packets. For other types of packets, the CID will be used to recover the constant fields. In the case of SEMICOMPRESSED_TCP packets, delta fields will be recorded. For COMPRESSED_TCP, delta fields will be calculated and recorded.

In [2], one more mechanism is designed to repair the de-synchronization between the compressor and the decompressor, which we will refer to as the "twice" algorithm. When a packet cannot be decompressed correctly, the decompressor will assume the reason is that one or more previous packets are lost. The decompressor also assumes that all the packets carry the same delta values and then tries to decompress the packets using these values two or more times. The twice algorithm improves the performance of header compression in certain circumstances. This algorithm has been developed as a proposed standard in IETF [3].

Usually, wireless links are prone to experience significant error rates and large delay times, which puts more difficulties on header compression. The Robust Header Compression (rohc) working group was created in IETF to improve header compression performance under this situation. In [5], some mechanisms are summarized for robust header compression over links with significant error rates and long round-trip times. One of the encoding methods, window_based LSB, is improved and presented as TCP_Aware RObust Header Compression (TAROC) in [6]. Window_based LSB [5] is an encoding method for delta values. If the compressor can make sure that the decompressor has received a group of packets, it can then transfer the delta values as the differences from that of the group, which are still expected to occupy fewer bits. TAROC uses a feature of TCP congestion control to decide which packets have already been received. The TCP sender keeps a sliding window. A new packet cannot be transferred without getting all the acknowledgments for the previous window. So the compressor tries to track the size of the sliding window according to the arriving sequence of packets. It will then know which the packets have been received by the decompressor and use window_based LSB accordingly.

3 New Challenges

A high frequency of some problems for computer networking was shown recently through tracing network connections [1] [11]. Header compression should be studied under the existence of these problems, since these problems could be encountered by users under typical operating conditions.

3.1 Packet Reordering

Packet reordering is not a new problem in computer networking. Unlike circuit-switching networks, where all the signals of one connection go through and wholly occupy a predefined route, the information may go through different routes in computer communication. In computer networks, the information that needs to be transferred between two ends is usually packaged into multiple packets. Each packet carries the address of the destination and enters the network. The routers will then try to deliver the packet according to the address as well as the current availability of network resources. So, packets of one connection are not guaranteed to all use the same route, which makes the sequence of arrival at the destination unknown. The usual belief is that when some network components are not working correctly, or there are changes to the network configuration, routers will transfer packets through different paths, which will cause packet reordering at the destination. This reasoning leads to the conclusion that the frequency and magnitude of the phenomenon are not severe.

Recently, Bennett et al. showed that reordering is not necessarily a rare occurrence in the Internet [1]. Even when a consistent path is chosen between a source and destination, the existence of multiple or redundant paths between routing neighbors, or within the data path of a single router, can cause packet reordering. In order to handle high speed networking, more and more multiple paths are being used between routing neighbors to form a logic link. At the same time, routers are also implementing more and more parallelism

internally, which includes parallel data processing paths and parallel decision making for packet forwarding. When packets belonging to one connection are coming in rapidly or the router's load is high, the packets can take any path through the routers and then reordering is unavoidable. Packet reordering is the natural result of the logic link as well as the router parallelism. Making changes to maintain the packet sequence within each connection might decrease the packet exchange speed. And keeping a high routing speed while eliminating packet reordering becomes quite expensive or even very difficult theoretically.

This non-pathological cause of reordering makes it more prevalent than previously believed. Reordering caused by the selection of different routes for packets associated with a particular connection may not be severe enough to cause significant problems in header compression. However, frequent and distant packet reordering caused within one route challenges header compression algorithms. One could argue that non-pathological packet reordering is not severe if the data transmission is really slow for a connection. This could be true under certain situations. But, as long as the bandwidth-delay product is large, the sliding window size of the TCP sender is large. It is possible that the sender has many packets to send and it can send as many as the window size out quite rapidly if the link connecting to the sender is not slow. Then these packets do have a high probability to be reordered. Some header compression algorithms assume and use the feature that the packets arrive with the same order as sent by the sender or with minor reordering. These algorithms need to be evaluated carefully considering more frequent packet reordering.

3.2 Errors passing CRC

When TCP/IP datagrams are passed over Ethernet, the link layer uses Cyclic Redundancy Check (CRC) to detect errors. PPP also uses CRC. In most wireless data networks, CRC is also the way to detect errors at the link layer. It has long been known that CRCs are very powerful for error detection. CRCs are based on polynomial arithmetic, base 2. CRC-32 is the most commonly used CRC in the TCP/IP suite. CRC-32 can detect all bursty errors with length less than 32 bits and all 2-bit errors less than 2048 bits apart. For all other types of errors, the chance of not detecting is just 1 in 2^{32}. Given this point, one can argue that the TCP or UDP checksum is not necessary. Practically, this was tried in the 1980s [11]. For some Network File Server (NFS) implementations, UDP checksum was disabled based on this argument. But this idea resulted in file corruption and ultimately was discarded as a bad idea.

Recently, Stone and Patridge have shown that for the Internet today, there are a wide variety of error sources which cannot be detected by link-level CRCs [11]. Defective hardware, buggy software, and problems in both end-systems and routers can all cause errors that will pass link-level error detection. Newly emerging software and hardware are especially vulnerable to these problems. In essence, any errors introduced at protocol layers above the link layer will not be detected by the link-layer CRC. Changes should be made to eliminate these error sources. But, before old error sources are cleaned up, or after new sources are introduced, the best solution is to detect these errors. In TCP, this task can be done only by Internet checksum [8] [9]. When errors pass the checksum, these errors will be passed to higher layers, including the application layer. The performance of header compression algorithms needs reconsideration under these errors.

4 Performance of Header Compression Algorithms Under New Challenges

For each header compression algorithm we can analyze the probability that the packets can not be decompressed, including corrupted packets and packets influenced by error propagation. Assume every bit has the same probability to be corrupted and every bit error will be detected by the error detection mechanism, either by the link layer or by the transport layer. Also, assume all the packets are of the same length. Then all the packets with the original header will have the same error probability, define it to be q_o. And all the

packets with the compressed header will have the same error probability and define it to be q_c. Since the compressed header is shorter than the original header, $q_c < q_o$. We will make the analysis using the length of an UNCOMPRESSED_TCP packet as 612 bytes and the length of a COMPRESSED_TCP packet as 517 bytes. The analysis for existing header compression algorithms is described as follows.

4.1 VJ compression

The problem with VJ header compression basically comes from the packet error or loss caused by error propagation. Since TCP uses the ACKs from the receiver as the congestion indication, the packet error or loss has more influence on TCP performance than just the erroneous/lost packet itself. This issue was discussed clearly in [2]. We give the packet error probability for VJ header compression in Figure 1, when the bit error rate (BER) is 10^{-5}. A corrupted COMPRESSED_TCP packet will be dropped. A COMPRESSED_TCP packet transferred correctly will also be discarded if the previous packet was dropped. So the packet error probability increases almost linearly until an UNCOMPRESSED_TCP packet is received. After that, the packet error probability will drop and start increasing again.

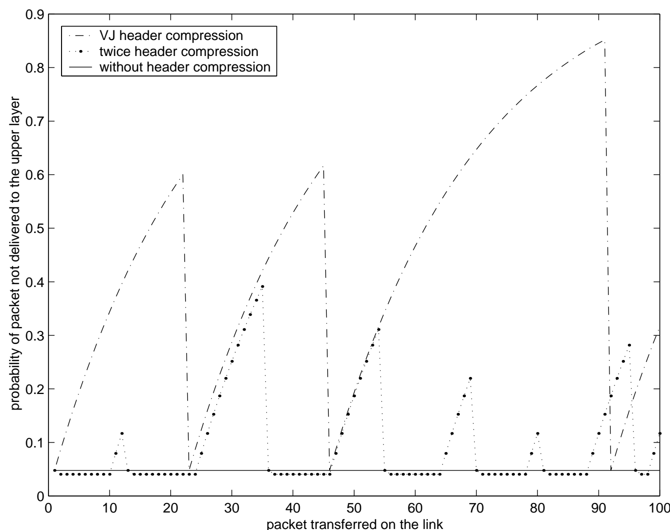

Figure 1: Packet error probability comparison, $BER = 10^{-5}$

Moreover, the compression efficiency is influenced by packet reordering. In the VJ compression algorithm, uncompressed reference packets are sent when a new CID is created, when constant fields change for a CID, and when the decompressor goes out of synchronization. When a packet can not be decompressed, TCP will detect it by timeout or duplicated ACKs and retransmit it. The compressor will find this retransmitted packet has a smaller sequence number than the previous packet, which causes a a negative delta value. The compression algorithm will disable the occurrence of negative delta value and force a full header to be transferred in this case. This mechanism makes sure that the information of one CID are refreshed on the decompressor side. But, it is also activated by packet reordering even when no packet loss occurs. When reordering happens, the sequence of packets arriving at the compressor is not the same as when they left the sender. There are packets that arrive at the compressor later than some packets which, at the TCP sender, enter the network following these packets and have larger sequence numbers. These packets will be sent with a full header instead of a compressed header. This introduces difficulties on estimating compression ef-

ficiency. Let B be the header size before compression and A the average compressed header size, supposing all other factors except packet reordering have been considered. If reordering is only a pathological behavior, which is rare, the compression ratio can be estimated as $R_{estimated} = A/B$. But since reordering could be more prevalent, we need to consider the effect on the compression ratio. Define x to be the percentage of packets that are received after their subsequently transmitted packets. Then, the compression ratio should be

$$R_{actual} = (A(1-x) + Bx)/B \tag{1}$$

Now, let's analyze $(R_{estimated} - R_{actual})/R_{actual}$, the error rate of the estimated compression ratio.

$$(R_{estimated} - R_{actual})/R_{actual} = 1 - 1/(1 + (B/A - 1)x) \tag{2}$$

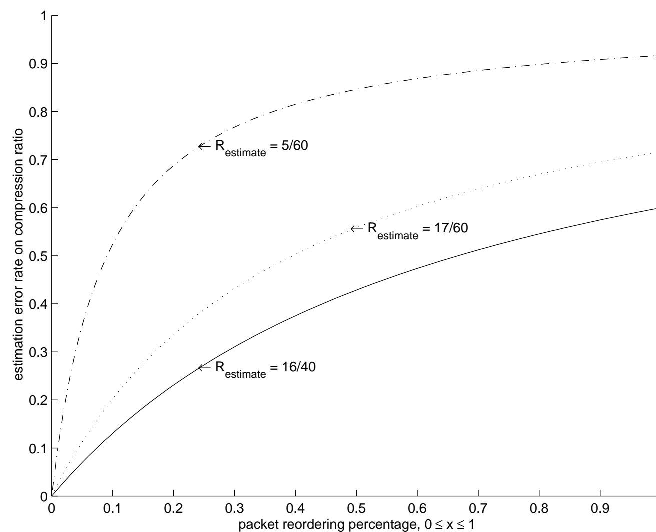

Figure 2: Estimation error rate of the compression ratio

This ratio is plotted in Figure 2. While using VJ compression, the typical size of the COMPRESSED_TCP is ranged from 4 to 17 bytes when no optional fields exist for the original headers. We show three curves for the estimation error rate on compression ratio. One is when the designed compression ratio is low, at 16 to 40. This corresponds to a TCP (20 bytes) and an IPv4 header (20 bytes) being compressed to 16 bytes. When packets need to be tunneled to a mobile node, the original packet is encapsulated in one extra IPv4 header. With one layer of encapsulation, the total header will be 60 bytes. It can be compressed to 17 bytes, which is shown in the second curve. It can also be compressed to 5 bytes and this is drawn in the third curve. We can find that the more efficient the compression algorithm, which is indicated by a smaller value of A/B, the more error on estimating the compression ratio given a certain value of x.

4.2 Twice Algorithm

Packet reordering also triggers the twice algorithm to send UNCOMPRESSED_TCP packets. This will decrease the compression ratio, the same as discussed for VJ compression.

Packet reordering has more influence on the performance of the twice algorithm. In this algorithm, if one packet is lost, subsequent packets are tried to be decompressed by applying their delta values two or

even more times. If packets are of the same size and there is no reordering/loss before the compressor, the algorithm will always recover the loss on the link between the compressor and decompressor. But, when there is reordering before the compressor, the result will be different.

Consider several packets belonging to one connection. We number them from 1 to 6, as the sequence when they were sent out from the sender. Assume the TCP data payload of the packets are all of the same length, d. Then, the compressed packets will all have the delta value d for sequence number.

$packet$	1	2	3	4	5	6
$deltavalue$	/	d	d	d	d	d

No matter which one is lost on the link, or even if several consecutive ones are corrupted, the decompressor can recover all other packets after the loss.

But, if packet 4 and 5 are reordered before the compressor, the delta value carried by the compressed packets are quite different and the twice algorithm can not work correctly under some cases. If packet 3 was corrupted, packet 5 can not be decompressed. The reason is the delta value carried by packet 5 is $2d$, applying it two or more times still can not recover the original value. Similarly, if (4), (5, 4), or (3, 5, 4) are lost, 6 can not be decompressed. Obviously, when packets are not of the same size, the twice algorithm also fails to eliminate error propagation.

$packet$	1	2	3	5	4	6
$deltavalue$	/	d	d	$2d$	$full-header$	$2d$

The reason for this error propagation phenomenon is that the twice algorithm assumes that most compressed packets have the same delta value for some applications. But, even for the these applications, reordered packets make this assumption invalid, which causes the twice algorithm not perform well. So if a packet cannot be decompressed, also using the packet length instead of the delta value to recover the sequence number could be helpful.

When a packet can not be decompressed, the twice algorithm assumes one or more packets are lost on the link between compressor and decompressor. But, packet reordering makes the sequence between packets more complex. Also, take the above example. This time, packet 6 comes before packets 4 and 5.

$packet$	1	2	3	6	4	5
$deltavalue$	/	d	d	$3d$	$full-header$	d

If packet 4 is lost, then packet 5 can not be decompressed using twice. Now, packet 5 seems to be coming later than the previous packet seen by the decompressor, instead of one packet missing before packet 5. In this situation, extending the twice algorithm to also subtract the delta value from the sequence number will be helpful.

In the examples we used, TCP/IP is assumed to have packet reordering only within a small range, up to 3 packets. It is clear that packet reordering does hamper the performance of the twice algorithm even in this situation.

The packet error probability for the twice algorithm is also shown in Figure 1. the twice algorithm works fine if most of the packets are of the same size and no packet reordering happened. When packet reordering does occur, the packet error probability could increase almost linearly just as in VJ header compression. If applying delta values more times can decompress a subsequent packet, the error probability will become low again. Otherwise, the probability can drop only when a packet with full header is received by the decompressor.

The errors that can pass link layer detection also influence the performance of the twice algorithm. We define *error masking probability* as the probability that those errors are not detected by the TCP checksum

and thus passed to higher layer. The error masking probability of TCP checksum needs more analysis. The influence on VJ header compression of this issue also needs further consideration. But, one point is clear, the error masking probability of the twice algorithm is higher than that of VJ compression. If the delta values are applied once and the TCP checksum is not correct, the actual reason may be some errors in the data payload or other bytes in the header of the packet. Then, when the delta values are applied two or even more times, the probability increases that the TCP checksum matches the packet. In this case, errors that could be found in the VJ algorithm will be passed to higher layers. For example, assume the decompressor is synchronized with the compressor and the previous sequence number is xxxxxxxx,xxxxxxxx,xxxxxx1x,xxxxxxxx, where x means either 0 or 1. And assume the decompressor receives a compressed packet with the delta value of sequence number 512, the delta value of ACK number 0. But in this packet, there is an error that avoided the link-level CRC. That is, one of the 2-octets as is used for TCP checksum, either in the header or in the data payload part, was changed from xxxxxxxx,xxxxxxxx,xxxxx1xx,xxxxxxxx to xxxxxxxx,xxxxxxxx,xxxxx0xx,xxxxxxxx. And only this one bit got corrupted. Then, when the decompressor tries to recover the packet, the TCP checksum indicates an error if the delta value is applied once. The TCP checksum is correct when the delta value is used twice, which is an indication that the packet has been decompressed correctly. Thus, the error is passed to the higher layer with a wrong sequence number.

Given that an error has passed the link-level CRC, the more times the delta value is applied, the more likely that the error can be passed to the layer above TCP. Similarly, if a sophisticated implementation of the twice algorithm [3] is used to make more educated guesses for the acknowledgment stream, where the delta value for the ACK number is not regular due to delayed ACK mechanism, there is an increasing probability that the errors can pass the transport layer error detection.

4.3 TCP_Aware RObust Header Compression (TAROC)

TCP_Aware RObust Header Compression (TAROC) improved one of the encoding methods, window_based LSB. W_LSB sends the delta fields only as the changed lower bits relative to the corresponding fields belonging to a certain window. TAROC restricts the size of the window according to the sliding window of TCP flow control. The TCP sender can transmit a packet only after all the packets belonging to the previous sliding window have been acknowledged by the receiver. In other words, if the window size is $cwnd$ when one packet is transmitted, the decompressor must have received correctly all those packets $cwnd$ before this packet. TAROC then use this knowledge and W_LSB encoding to compress the header. Theoretically, TAROC will work perfectly if the compressor knows when the sender changes the window size. Unfortunately, this is not an easy task. TAROC uses the packet arrival sequence as the window size indication. Again, packet reordering will influence the accuracy of this estimate. If packet reordering will be assumed as an indication of packet loss at the compressor, the compressor believes that the sender has adopted a smaller window size following that. Thus, the decompressor is assumed to has received a number of packets correctly, where the number is more than what actually has been received. This will potentially make the decompressor work incorrectly. We give the packet error probability in Figure 3. In order to show the difference between TAROC and TCP/IP without header compression, we use a logarithmic scale on the Y-axis. The probability for TAROC is very low if the compressor can make an accurate estimate of the sliding window size. However, if the compressor makes a smaller estimation, all the packets will be dropped until an UNCOMPRESSED_TCP packet is received.

On the other hand, for some TCP connections with large bandwidth-delay products, the sliding window size could be very large. The advantage of W_LSB will become smaller or even no bits can be compressed in this situation.

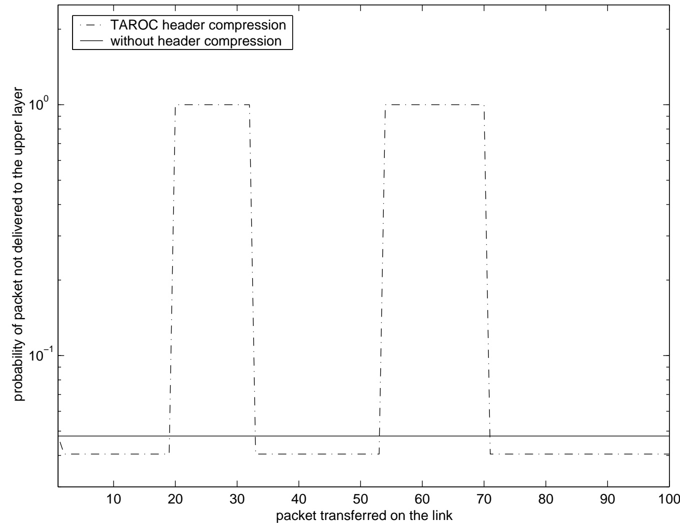

Figure 3: Packet error probability comparison, $BER = 10^{-5}$

5 Adaptive Header Compression Algorithm

5.1 Wireless Link and Connection Consideration

For mobile radio channels, especially the channels used for typical transmissions from a base station to mobile users in an urban environment, Rayleigh fading is the widely accepted model. Fading causes periods of significant degradation of the received signals. Received signals determine the bit error rate (BER) seen by the receiver. When the signal to noise ratio is above a certain threshold, the channel is assumed to be in the *good* state with a low BER. On the other hand, when the ratio is below this threshold, the channel is in the *bad* state, where the BER is quite high. Under the Rayleigh fading model, the wireless channel is considered to have these two states and keep changing between the states.

When the wireless channel is in the *good* state, original packets can be transferred correctly with high probability. Compressed packets can have an even higher probability of correct transmission. Any saving on the bandwidth will be beneficial for the particular connection as well as the whole network. The only task in this state is to lessen or eliminate error propagation. When the wireless channel is in the *bad* state, both original packets and compressed packets can barely be transmitted correctly. As long as the header compression algorithm can lessen the error propagation for the next *good* period, the algorithm will have good performance. Unfortunately, it is almost impossible for the compressor of a wireless channel to know the current state of the link. But the sender may know some general characteristics of the wireless link and select a proper header compression algorithm. The algorithm can also be adapted following the channel changes throughout the connection period.

From the performance discussion of previous header compression algorithms, we can reach the conclusion that assuming no reordering or minor reordering will introduce problems in typical connections, especially those with large bandwidth-delay products. While the influence of packet reordering on TCP performance needs more consideration, the best use of the communication channel is to transfer these packets efficiently. So the task of header compression becomes transferring the packets in the sequence as received from the previous node, using as little bandwidth as possible.

5.2 Adaptive Header Compression

Based on the previous discussion, we present the following algorithm, the adaptive header compression algorithm, to achieve a good tradeoff between the throughput and the compression ratio.

The packet types we use are similar to those defined in [7] [3] [5] [6]. We use the W_LSB to do the delta field encoding. We can also combine all the delta fields together and use padding to keep them aligned to byte boundaries. For each packet, we use a certain number of packets received before hand to do the W_LSB encoding. Since assuming a fixed sequence between the TCP packets will potentially hamper the performance, we make no such assumption. As long as the decompressor has received any packet of this group correctly, it will decompress this packet correctly. For wireless links in the *good* state or if the period of *bad* state is short, the probability that a group of packets are all corrupted is still low. In a long-term *bad* state, the probability could be high. In order to handle this, we send UNCOMPRESSED_TCP packets periodically to refresh the decompressor. By designing the algorithm carefully, we can send more packets during the *good* state while making decompressor work properly shortly after the *bad* state ends.

The compressor performs as follows:

- Define two variables, $windowSize$ and $distance$. These two variables can be adapted according to the link condition and the average packet size. $windowSize$ determines how many packets should be used to do W_LSB encoding, and $distant$ defines how frequently a SEMICOMPRESSED_TCP should be sent to make the decompressor synchronized with the compressor.

- For the first $windowSize$ packets belonging to the connection, send an UNCOMPRESSED_TCP packet with exponential distance, send other packets as SEMICOMPRESSED_TCP. In other words, send packet $(2^i - 1)$ as UNCOMPRESSED_TCP and all other packets as SEMICOMPRESSED_TCP. If the $windowSize$ is small, send all the first $windowSize$ packets as UNCOMPRESSED_TCP.

- After this, send a SEMICOMPRESSED_TCP packet every $distance$ packets. If within the previous $distance$ packets one SEMICOMPRESSED_TCP has been sent, send a COMPRESSED_TCP instead.

- Whenever one or more constant fields change, send the packet as UNCOMPRESSED_TCP.

- If a packet has a sequence number seen by the compressor before, send it as UNCOMPRESSED_TCP.

- Other packets are sent as COMPRESSED_TCP, with delta fields using W_LSB encoding. The window is composed of the previous $windowSize$ packets sent by the compressor.

When the packet size is large, or there is severe packet reordering, or $windowSize$ is big, COMPRESSED_TCP sometimes can not compress on the delta fields. If this happens, as described for sending SEMICOMPRESSED_TCP, we will skip the next SEMICOMPRESSED_TCP. This will increase the compression ratio without increasing error propagation.

For the decompressor:

- Record or update the fields when receiving an UNCOMPRESSED_TCP packet and pass the packet to the upper layer.

- Recover the constant fields in a SEMICOMPRESSED_TCP packet and update the delta fields. If there is no UNCOMPRESSED_TCP received before this packet, store this packet and decompress it whenever an UNCOMPRESSED_TCP packet arrives.

- Recover the COMPRESSED_TCP packets based on the most recent received packet. If the decompressed packet is correct as indicated by the TCP checksum, update the delta fields.

From the compression method, it is clear that packet reordering will not make the compression malfunction. The only influence is that the length of delta values might be longer. The value of $windowSize$ and $distance$ are determined by the average BER of the link and the average packet size.

The higher the BER, the bigger the $windowSize$ and the smaller the $distance$. On the other hand, the lower the BER, the smaller the $windowSize$ and the bigger the $distance$. When $windowSize = \infty$ or $distance = 1$, all the packets are sent as SEMICOMPRESSED_TCP, which is for links with a very high error rate. When $windowSize$ becomes smaller, the COMPRESSED_TCP contains a shorter header, which will increase the compression ratio. But as the probability that all the packets in the $windowSize$ got corrupted becomes high, the error propagation becomes severe. One thing to be noticed is that for a certain BER level and average packet length, there exists a proper $windowSize$ value. This $windowSize$ value will give good performance and increasing this value will not make much difference in the packet error rate. As $distance$ grows bigger, fewer SEMICOMPRESSED_TCP packets will be sent, which is suitable for links with lower BER. When $distance$ decreases, more frequent SEMICOMPRESSED_TCP packets will refresh the state of the decompressor more often. This will give good decompression results even for higher BER.

Both $windowSize$ and $distance$ are influenced by the average packet size. For small packets, we can select a large $windowSize$ and a large $distance$. A large $windowSize$ decreases the error propagation, which can still yield a good compression ratio for small packets. Since the error propagation has been decreased by large $windowSize$, we can then choose a larger $distance$, which in turn increases the compression ratio. Similarly, we can use small values of $windowSize$ and $distance$ for large packets.

5.3 Performance of Adaptive Header Compression

For our Adaptive Header Compression algorithm, define the error probability for the SEMICOMPRESSED_TCP to be q_s and let p_i be the error probability for packet i. Then, assuming no UNCOMPRESSED_TCP header is generated after the first $windowSize$ packets, the error probability of every packet is as follows:

$p_i = q_o$ where $i < windowSize$ and $i = 2^j - 1, j = 1, 2, \cdots$

$p_i = q_s$ where $i < windowSize$ and $i \neq 2^j - 1, j = 1, 2, \cdots$

$p_i = q_s$ where $i \geq windowSize$ and $\frac{i - windowSize}{distance} = j, j = 0, 1, \cdots$

$p_i = q_c + \prod_{k=1}^{windowSize} q_{i-k} \times (1 - q_c)$ where $i \geq windowSize$ and $\frac{i - windowSize}{distance} \neq j, j = 0, 1, \cdots$

We give the calculated packet error probability in Figure 4 and Figure 5. These figures demonstrate the error probability for each packet sent from the compressor. We use a bit error rate of 10^{-5}, the length of an UNCOMPRESSED_TCP packet as 612 bytes, the length of a SEMICOMPRESSED_TCP packet as 523 bytes, and the length of a COMPRESSED_TCP packet as 517 bytes. We also assume no constant fields of the header have changed. In other words, all packets are sent using the compressed header when possible. The packet error probabilities are plotted in Figure 1 and Figure 3 when VJ, twice, and TAROC are used under the same conditions. Using the algorithm we proposed the packet error rates are either p_o or p_s for the first $windowSize$ packets. After that, the error rate oscillates within a range. Periodically, a packet with original delta fields will be sent, which has a higher error probability. Between these packets, packets are sent using the compressed header, with almost a constant lower error probability. Figure 4 presents the situation when $distance$ is selected as 4. Figure 5 is the situation with $distance$ as 16.

It is clear that the average packet error probability is even smaller when $distance$ becomes larger within a certain range. This seems to conflict with the design objective. However, recall that the formula was computed using the assumption that the errors are uniformly distributed. The graphs thus indicate the situation when the wireless link is in *good* state. When the wireless link just exits a *bad* state, a smaller $distance$ will refresh the state of the decompressor quicker and give better decompression results.

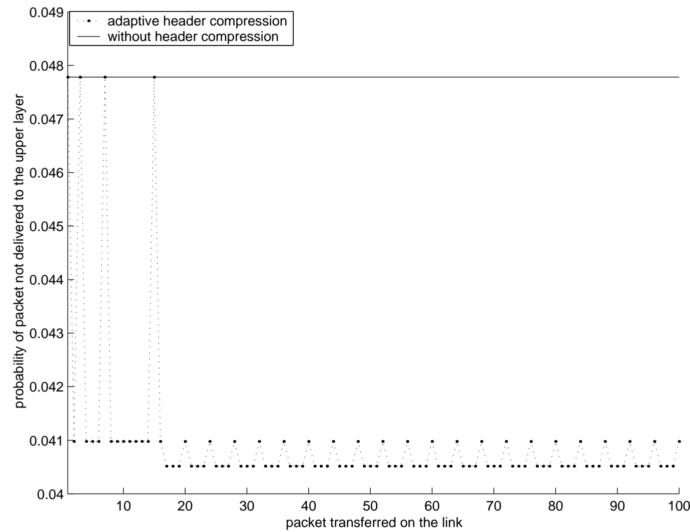

Figure 4: Packet error probability using Adaptive compression, $windowSize = 16$, $distance = 4$, $BER = 10^{-5}$

6 Conclusion

So far, we have discussed some recently observed computer network problems. These problems worsen the performance of existing header compression algorithms, which are also described. Moreover, we provide a new algorithm, which can achieve better performance when used over wireless links and can address these problems. Our further work includes using simulation or emulation to test how packet reordering and packet loss influence the performance of header compression schemes for application traffic.

References

[1] J. Bennett, C. Partridge, and N. Schectman. Packet reordering is not pathological network behavior, 1999.

[2] M. Degermark, M. Engan, B. Nordgren, and S. Pink. Low-loss TCP/IP header compression for wireless networks. In *MobiCom*, 1996.

[3] M. Degermark, B. Nordgren, and S. Pink. IP header compression. RFC 2507, Internet Engineering Task Force, February 1999.

[4] S. Derring and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Internet Engineering Task Force, December 1998.

[5] C. Bormann (ed.) *et al.* RObust Header Compression (ROHC). Internet Draft (work in prograss) draft-ietf-rohc-rtp-09.txt, Internet Engineering Task Force, February 2001.

[6] H. Liao *et al.* TCP-Aware RObust Header Compression (TAROC). Internet Draft (work in prograss) draft-ietf-rohc-tcp-taroc-01.txt, Internet Engineering Task Force, March 2001.

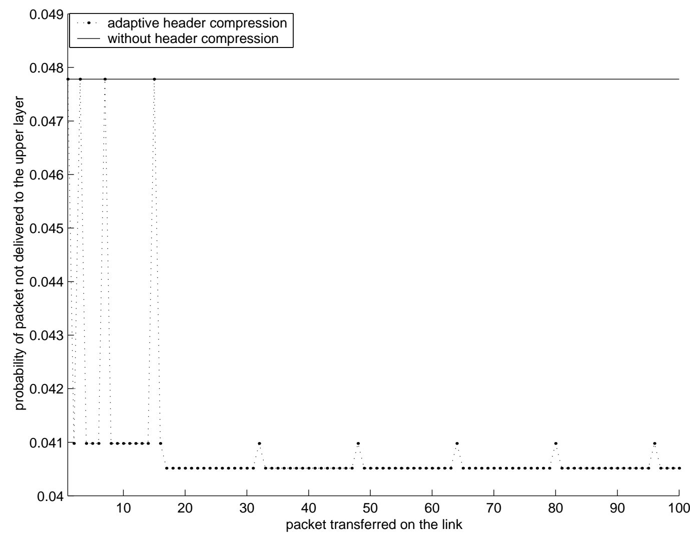

Figure 5: Packet error probability using Adaptive compression, $windowSize = 16$, $distance = 16$, $BER = 10^{-5}$

[7] V. Jacobson. IP headers for low-speed serial links. RFC 1144, Internet Engineering Task Force, February 1990.

[8] J. Postel. Transmission Control Protocol. RFC 793, Internet Engineering Task Force, September 1981.

[9] A. Rijsinghani. Computation of the Internet Checksum via Incremental Update. RFC 1624, Internet Engineering Task Force, May 1994.

[10] A. K. Salkintzis. A Survey of Mobile Data Networks. *IEEE Communications Surveys*, pages Vol 2, No.3, Third Quarter, 1999.

[11] J. Stone and C. Partridge. When The CRC and TCP Checksum Disagree. *ACM SIGCOMM*, September 2000.