Low-loss TCP/IP Header Compression for Wireless Networks

Mikael Degermark[†], Mathias Engan[†], Björn Nordgren[†], and Stephen Pink[†‡]

{micke, engan, bcn, steve}@cdt.luth.se

[†]CDT/Department of Computer Science
Luleå University
S-971 87 Luleå, Sweden

[‡]Swedish Institute of Computer Science
PO box 1263
S-164 28 Kista, Sweden

Abstract

Wireless is becoming a popular way to connect mobile computers to the Internet and other networks. The bandwidth of wireless links will probably always be limited due to properties of the physical medium and regulatory limits on the use of frequencies for radio communication. Therefore, it is necessary for network protocols to utilize the available bandwidth efficiently.

Headers of IP packets are growing and the bandwidth required for transmitting headers is increasing. With the coming of IPv6 the address size increases from 4 to 16 bytes and the basic IP header increases from 20 to 40 bytes. Moreover, most mobility schemes tunnel packets addressed to mobile hosts by adding an extra IP header or extra routing information, typically increasing the size of TCP/IPv4 headers to 60 bytes and TCP/IPv6 headers to 100 bytes.

In this paper, we provide new header compression schemes for UDP/IP and TCP/IP protocols. We show how to reduce the size of UDP/IP headers by an order of magnitude, down to four to five bytes. Our method works over simplex links, lossy links, multi-access links, and supports multicast communication. We also show how to generalize the most commonly used method for header compression for TCP/IPv4, developed by Van Jacobson, to IPv6 and multiple IP headers. The resulting scheme unfortunately reduces *TCP throughput over lossy links due to unfavorable interaction with TCP's congestion control mechanisms. However, by adding two simple mechanisms the potential gain from header compression can be realized over lossy wireless networks as well as point-to-point modem links.*

[*]This work was supported by grants from the Centre for Distance Spanning Technology (CDT), Luleå, Sweden, and Ericsson Radio Systems AB.

1 Introduction

An increasing number of end-systems are being connected to the global communication infrastructure over relatively low-speed wireless links. This trend is largely driven by users that carry their computers around and need a convenient way to connect to the Internet or other networks. In the core of the global communication infrastructure, optic fibers provide high speeds, high reliability and low bit-error rates. But an increasing number of first and last hops in the network are using wireless technology with limited bandwidth, intermittent connectivity, and relatively high bit-error rates. The TCP/IP protocol suite needs to be augmented to accommodate this type of link and need mechanisms to utilize them efficiently.

In the local area, several commercial wireless LAN technologies offer wireless communication at speeds of 1-2 Mbit/s. Infrared technologies provide similar speeds. In the wide area, several cellular phone technologies offers data channels with speeds of a few kbit/s, for example the European GSM at 9600 bit/s and CDPD at 19.2 kbit/s. Even though there are plans to increase bandwidth, in the foreseeable future it is likely that wireless bandwidth, especially in the wide area and outside population centers, will be a scarce resource due to properties of the physical medium and regulatory limitations on the use of radio frequencies.

Mobile users on wireless networks will want the same services as they already have when using stationary computers attached to the wired Internet. Therefore it is important to utilize the limited bandwidth over wireless links efficiently. However, two trends threaten to decrease the efficiency of Internet technology over wireless links. The first is the coming of the next generation of the Internet Protocol, IPv6. With IPv6 the address size increases from 4 bytes to 16 bytes, and the basic IP header from 20 bytes to 40 bytes. In addition, various extension headers can be added to the basic IPv6 header to provide extra routing information, authentication, etc. IPv6 with its large headers is clearly intended for networks where there is plenty of bandwidth and packets are large so that the header overhead is negligible.

The second trend is mobility. There are several schemes for allowing a host to keep its original IP address even though it has moved to a different part of the network. These schemes usually involve a *home agent* in the home subnet to capture packets addressed to the mobile computer and *tunnel* them to where the mobile computer happens to be attached.

Tunneling is done by encapsulating the original packet with an extra IP header. With one level of encapsulation the minimal header of a TCP segment is 100 bytes[1]. In the latest proposal for Mobile IPv6, the mobile host can inform its correspondents about its current location. This allows correspondents to optimize the route by not visiting the home network. Correspondents add a one-address routing header to the basic IPv6 header, adding 24 bytes to the header for a total of 84 bytes for a TCP segment. This procedure increases the header size over the first hop, where it would otherwise be 60 bytes, and decreases it over the last hop. In the latest proposal for mobile IPv6, all headers are transferred over the wireless links. While the mobility protocols are essential for convenient attachment of mobile computers to the Internet, the large headers are detrimental when bandwidth is limited.

In this paper we show how large headers of 50 bytes or more can be reduced in size to 4-5 bytes. The efficiency of our scheme is based on there being consecutive headers belonging to the same packet stream that are identical or changes seldom during the life of the packet stream. This allows the upstream node to send a short index identifying a previously sent header stored as state in the downstream node instead of sending the complete header. Header compression has several important benefits for the user:

1. When packets contain little data the overhead of large headers can cause unacceptable delays. For TELNET, a typical packet contains one byte of data. The minimum IPv6/TCP header is 60 bytes, adding an encapsulating IP header for mobility increases the header size to 100 bytes. Transmitting this header over a 9600 bit/s GSM link takes 84 ms resulting in a round-trip time (for the echoed character) of at least 168 ms. This results in too long response times, around 100 ms is acceptable, and the system will appear sluggish. By reducing the header to 4-5 bytes the round-trip time over the GSM link can be reduced to less than 10 ms which allows for queuing and propagation delays in the rest of the path.

2. The overhead of large headers can be prohibitive when many small packets are sent over a link with limited bandwidth. The acceptable end-to-end delay budget when people talk to each other can be as low as 150 ms, depending on the situation. The propagation delay (due to the limited speed of light in a fiber) is ideally about 20 ms across USA and 100 ms to the farthest point in a global network. Since audio can have a relatively low data rate, around 10-14 kbit/s, the time required to fill a packet with audio samples is significant. To allow for queuing delay and end system processing it is necessary to use small packets that are filled quickly if the delay budget is to be met. However, sending more packets increase header overhead. Table 1 shows the bandwidth consumed by headers for various headers and times between packets. *Optim* means an IPv6/UDP header with a one-address routing header; used for example in Mobile IPv6 route optimization. *Tunnel* means an IPv6/UDP header encapsulated in an IPv6 header; used for example in Mobile IPv6. *Routing* means an IPv6/UDP header with a four address routing header. *Compr* means the compressed version of IPv6/UDP, *optim*, *tunnel*, or *routing*. For comparison, the bandwidth needed for the actual audio samples is somewhere between 10 kbit/s for GSM quality to 128 kbit/s for CD quality [13, p. 179]. So when tunneling for mobility, at least 45.2 kbit/s is required for GSM quality with 20 ms between packets. With header compression this can be reduced to 11.6 kbit/s.

	Header bw, kbit/s		
Pkt interval	80 ms	40 ms	20 ms
IPv4/UDP	2.8	5.6	11.2
IPv6/UDP	4.8	9.6	19.2
optim	7.2	14.4	28.8
tunnel	8.8	17.6	35.2
routing	12.0	24.0	48.0
compr (4 byte)	0.4	0.8	1.6

Table 1: Required bandwidth for headers, kbit/s

3. TCP bulk transfers over the wide area today typically use 512 byte segments. With tunneling, the TCP/IPv6 header is 100 bytes. Reducing the header to 5 bytes reduces the overhead from 19.5 per cent to less than one per cent, thus reducing the

[1]For IPv6. 60 bytes for IPv4.

total time required for the transfer. With smaller segments or larger headers[2] the benefit from header compression is even more pronounced.

An IPv6 node is required to perform path MTU[3] discovery when sending datagrams larger than 596 bytes because datagrams are not fragmented by the network in IPv6. A node could restrict itself to never send datagrams larger than 596 bytes, but it is likely that most transfers will use larger datagrams. If datagrams are 1500 bytes[4], header compression reduces header overhead from 7.1 per cent to 0.4 per cent.

4. Because fewer bits per packet are transmitted with header compression, the packet loss rate over lossy links is reduced. This results in higher quality of service for real-time traffic and higher throughput for TCP bulk transfers.

The structure of our paper is as follows. After providing motivation for header compression for IPv6, we describe our new soft-state-based header compression algorithm for UDP/IPv6, with its support for simplex streams, etc. We then show with simulation results that the traditional scheme for TCP/IP header compression does not work well over lossy-links such as wireless. We suggest additional mechanisms for improving performance on a high loss environment, and show their viability with simulation results. We then report on the implementation status of our header compression scheme and conclude with a section on related work and a summary.

2 Header compression

The key observation that allows efficient header compression is that in a *packet stream*, most fields are identical in headers of consecutive packets. For example, figure 1 show a UDP/IPv6 header with the fields expected to stay the same colored grey. As a first approximation, you may think of a packet stream as all packets sent from a particular source address and port to a particular destination address and port using the same transport protocol.

With this definition of packet stream, in figure 1 addresses and port numbers will clearly be the same in all packets belonging to the same stream. The IP version is 6 for IPv6 and the Next Hdr field will have the value representing UDP. If the Flow Label field is nonzero, the Prio field should by specification not change frequently . If the Flow Label field is zero, it is *possible* for the Prio field to change frequently, but if it does, the definition of what a packet stream is can be changed slightly so that

[2]An IPv6 routing header containing 24 addresses is 392 bytes long!

[3]The path MTU is the maximum size of packets transmitted over the path.

[4]The maximum size of Ethernet frames is 1500 bytes.

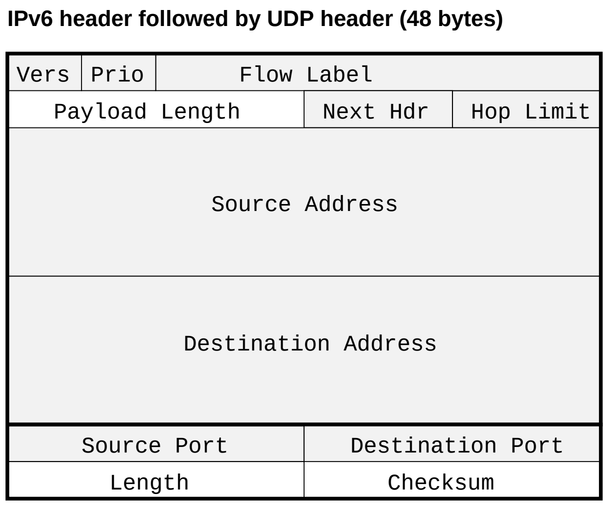

Figure 1: Unchanging fields of UDP/IPv6 packet.

packets with different values of the Prio field belong to different packet streams. The Hop Limit field is initialized to a fixed value at the sender and is decremented by one by each router forwarding the packet. Because packets usually follow the same path through the network, the value of the field will change only when routes change.

The Payload length and Length fields give the size of the packet in bytes. Those fields are not really needed since that information can be deduced from the size of the link-level frame carrying a packet, provided there is no padding of that frame.

The only remaining field is the UDP checksum. It covers the payload and the pseudo header, the latter consisting of the Nxt Hdr field, the addresses, the port numbers and the UDP Length. Because the checksum field is computed from the payload, it will change from packet to packet.

To compress the headers of a packet stream a compressor sends a packet with a *full header*, essentially a regular header establishing an association between the non-changing fields of the header and a *compression identifier*, CID, a small unique number also carried by compressed headers. The full header is stored as *compression state* by the decompressor. The CIDs in compressed headers are used to lookup the appropriate compression state to use for decompression. In a sense, all fields in the compression state is replaced by the CID. Figure 2 shows full and compressed headers. The size of a packet might be optimized for the MTU[5] of the link, to avoid increasing the packet size for full headers, the CID is carried in length fields. Full UDP headers also

[5]Maximum Transmission Unit, maximum size of packets transmitted over the link.

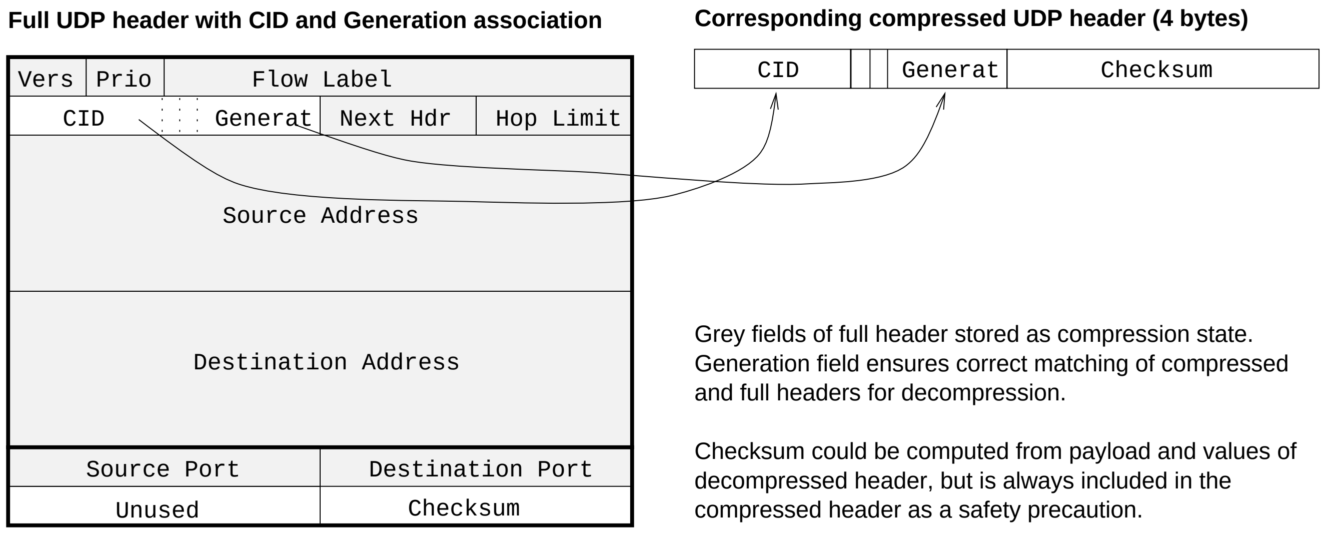
Full UDP header with CID and Generation association
Corresponding compressed UDP header (4 bytes)
Vers
Prio
Flow Label
CID
Generat
Next Hdr
Hop Limit
Source Address
Destination Address
Source Port
Destination Port
Unused
Checksum
CID
Generat
Checksum
Grey fields of full header stored as compression state. Generation field ensures correct matching of compressed and full headers for decompression.
Checksum could be computed from payload and values of decompressed header, but is always included in the compressed header as a safety precaution.

Figure 2: Full and compressed headers.

contain a generation field used for detection of obsolete compression state (see section 3).

All fields in headers can be classified into one of the following four categories depending on how they are expected to change between consecutive headers in a packet stream. [8] provides such classifications for IPv6 basic and extension headers, IPv4, TCP, and UDP headers.

nochange The field is not expected to change. Any change means that a full header must be sent to update the compression state.

inferred The field contains a value that can be inferred from other values, for example the size of the frame carrying the packet, and thus need not be included in compressed headers.

delta The field may change often but usually the difference from the field in the previous header is small, so that it is cheaper to send the change from the previous value rather than the current value. This type of compression is used for fields in TCP headers only.

random The field is included *as-is* in compressed headers, usually because it changes unpredictably.

Because a full header must be sent whenever there is a change in **nochange** fields, it is essential that packets are grouped into packet streams such that changes occur seldomly within each packet stream.

The compression method outlined above would work very well in the ideal case of a lossless link. In the real world bit-errors will result in lost packets and the loss of a full header can cause inconsistent compression state at compressor and decompressor, resulting in *incorrect decompression*, expanding headers to be different than they were before compressing them. A header compression method needs mechanisms to avoid incorrect decompression due to inconsistent compression state and it needs to update the compression state if it should become inconsistent. Our scheme use different mechanisms for UDP and TCP, covered in sections 3 and 4.

If header compression would result in significantly increased loss rates, the gains from the reduced header size could be less than the reduced throughput due to loss. All in all, header compression would then decrease throughput. In the following, we show how this can be avoided and the potential gain from header compression can be realized even over lossy links.

3 UDP header compression

For UDP packet streams the compressor will send full headers periodically to refresh the compression state. If not refreshed, the compression state is garbage collected away. This is an application of the *soft state* principle introduced by Clark [3] and used for example in the RSVP [19] resource reservation setup protocol, and the PIM [6] multicast routing protocol.

The periodic refreshes of soft state provide the following advantages.

- If the first full header is lost, the decompressor can install proper compression state when a refreshing header arrives. This is also true when there is a change in a **nochange** field and the resulting full header is lost.

- When a decompressor is temporarily disconnected from the compressor, a common situation for wireless, it can install proper compression state when the connection is resumed and a refresh header arrives.

- In multicast groups, periodic refreshes allow new receivers to install compression state without explicit communication with the compressor.

- The scheme can be used over simplex links as no upstream messages are necessary.

3.1 Header Generations

We do not use incremental encoding of any header fields that can be present in the header of a UDP packet. This means that loss of a compressed header will not invalidate the compression state. It is only loss of a full header that would change the compression state that can result in inconsistent compression state and incorrect decompression.

To avoid such incorrect decompression, each version of the compression state is associated with a *generation*, represented by a small number, carried by full headers that install or refresh that compression state and in headers that were compressed using it. Whenever the compression state changes, the generation number is incremented. This allows a decompressor to detect when its compression state is out of date by comparing its generation to the generation in compressed headers. When the compression state is out of date, the decompressor may drop or store packets until a full header installs proper compression state.

3.2 Compression Slow-Start

To avoid long periods of packet discard when full headers are lost, the refresh interval should be short. To get high compression rates, however, the refresh interval should be long. We use a new mechanism we call *compression slow-start* to achieve both these goals. The compressor starts with a very short interval between full headers, one packet with a compressed header, when compression begins and when a header changes. The refresh interval is then exponentially increased in size with each refresh until the steady state refresh period is reached. Figure 3 illustrates the slow-start mechanism, tall lines represents packets with full headers and short lines packets with compressed headers. If the first packet is lost, the compression state will be synchronized by the third packet and only a single packet with a compressed header must be discarded or stored temporarily. If the first three packets are lost, two additional packets must be discarded or stored, etc. We see that when the full header that updates the compression state after a change is lost in an error burst of x packets, at most $x-1$ packets are discarded or stored temporarily due to obsolete compression state.

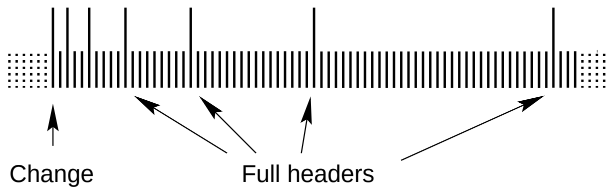

Figure 3: Compression slow-start after header change. All refresh headers carry the same generation number.

With the slow-start mechanism, choosing the interval between header refreshes becomes a tradeoff between the desired compression rate and how long it is acceptable to wait before packets start coming through after joining a multicast group or coming out from a radio shadow. We propose a time limit of at most 5 seconds between full headers and a maximum number of 256 compressed headers between full headers. These limits are approximately equal when packets are 20 ms apart.

3.3 Soft-state

We are able to get soft state by trading off some header compression. A hard-state based scheme does not send refresh messages and so will get more compression. The amount of compression lost in our soft state approach, however, is minimal. Figure 4 shows the average header size when full headers of size H are sent every xth packet, and the others have compressed headers of size C. For comparison, the diagram also shows the size of the compressed header. The values used for H and C are typical for UDP/IPv6. It is clear from figure 4 that if the header refresh frequency is increased past the knee of the curve, the size of the average header is very close to the size of the compressed header. For example, if we decide to send 256 compressed headers for every full header, roughly corresponding to a full header every five seconds when there are 20 ms between packets, the average header is 1.4 *bits* larger than the compressed header.

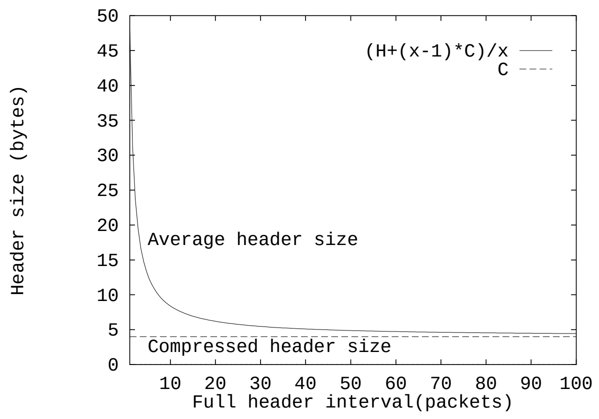

Figure 4: Average header size. $H = 48$, $C = 4$.

Figure 5 shows the bandwidth efficiency, i.e., the fraction of the consumed bandwidth used for actual data.

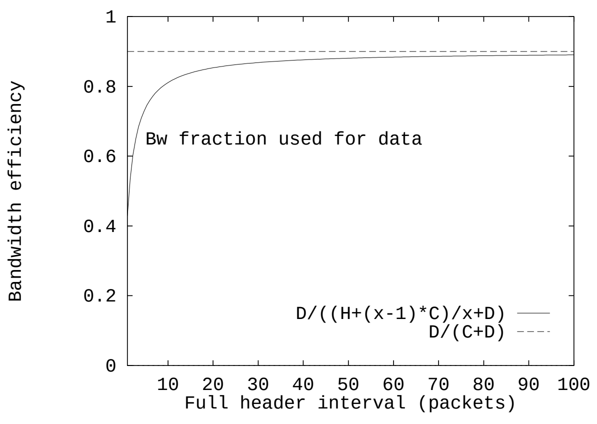

Figure 5: Bandwidth Efficiency. $H = 48$, $C = 4$, $D = 36$.

The bandwidth efficiency when all headers are compressed is shown for comparison. The size of the data, D, is 36 bytes, which corresponds to 20 ms of GSM encoded audio samples.

Figures 4 and 5 show that, when operating to the right of the knee of the curve, the *size* of the compressed header is more important than *how often* the occasional full header is sent due to soft state refreshes or changes in the header. The cost is slightly higher than for handshake-based schemes, but we think that is justified by the ability of our scheme to compress on simplex links and compress multicast packets on multi-access links.

3.4 Error-free compression state

Header compression may cause the error model for packet streams to change. Without header compression, a bit-error damages only the packet containing the bit-error. When header compression is used and bit-errors occur in a full header, a single error could cause loss of subsequent packets. This is because the bit-error might be stored as compression state and when subsequent headers are expanded using that compression state they will contain the same bit-error.

If the link-level framing protocol uses a strong checksum, this will never happen because frames with bit-errors will be discarded before reaching the decompressor. However, some framing protocols, for example SLIP [16], lack strong checksums. PPP[17] has a strong checksum if HDLC-like framing [18] is used, but that is not required.

IPv6 must not be operated over links that can deliver a significant fraction of corrupted packets. This means that when IPv6 is run over a lossy wireless link the link layer must have a strong checksum or error correction. Thus, the rest of this discussion about how to protect against bit-errors in the compression state is not applicable to IPv6. These mechanisms are justified only when used for protocols where a significant fraction of corrupted packets can be delivered to the compressor.

It is sufficient for compression state to be installed properly in the decompressor if one full header is transmitted undamaged over the link. What is needed is a way to detect bit-errors in full headers. The compressor extends the UDP checksum to cover the whole full header rather than just the *pseudo-header* since the pseudo-header doesn't cover all the fields in the IP header. The decompressor then performs the checksum before storing a header as compression state. In this manner erroneous compression state will not be installed in the decompressor and no headers will be expanded to contain bit-errors. The decompressor restores the original UDP checksum before passing the packet up to IP.

Once the compression state is installed, there will be no extra packet losses with UDP header compression. If the decompressor temporarily stores packets for which it does not have proper compression state and expands their headers when a matching full header arrives, there will be no packet loss related to header compression. The stored packets will be delayed, however, and hard real-time applications may not be able to utilize them, although adaptive applications might.

3.5 Reduced packet loss rate

Header compression reduces the number of bits that are transmitted over a link. So for a given bit-error rate the number of transmitted packets containing bit-errors is reduced by header compression. This implies that header compression will improve the quality of service over wireless links with high bit-error rates, especially when packets are small, so that the header is a significant fraction of the whole packet.

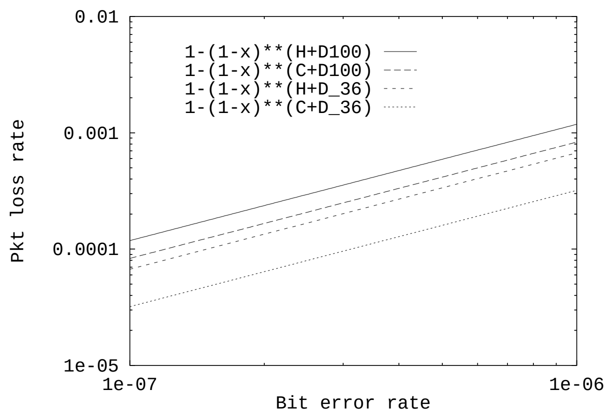

Figure 6: Packet loss rate as a function of bit-error rate, with and without header compression and for payloads of 36 and 100 bytes.

Figure 6 shows the packet loss rate as a function of the bit-error rate of the media with and without header compression. The packet loss rates for compressed packets assume that the compression state has been successfully installed. Compressed headers, C, are 4 bytes, full and regular headers, H, are 48 bytes (IPv6/UDP). D is the size of the payload.

Thus, our header compression scheme for UDP/IP in addition to decreasing the required header bandwidth, also reduces the rate of packet loss. The packet loss rate is decreased in direct proportion to the decrease in packet size due to header compression. For the 36 byte payload, the packet loss rate is decreased by 52% and for the 100 byte payload by 30%. With tunneling, the packet loss rate decreases by 68% and 45%, respectively.

If bit-errors occur in bursts whose length is of the same order as the packet size, there will be little or no improvement in the packet loss frequency because of header compression. The numbers above assume uniformly distributed bit-errors.

4 TCP header compression

The currently used header compression method for TCP/IPv4 is by Jacobson [10], and is known as VJ header compression. Jacobson carefully analyzes how the various fields in the TCP header change between consecutive packets in a TCP connection. Utilizing this knowledge, his method can reduce the size of a TCP/IPv4 header to 3–6 bytes.

It is straightforward to extend VJ header compression to TCP/IPv6. It is important to do this since not only are the base headers in IPv6 larger than IPv4, multiple headers needed to support Mobile IPv6[15], i.e., routing headers with 16 byte addresses tunneled to the mobile host, will produce a large overhead on wireless networks.

4.1 Compression of TCP header

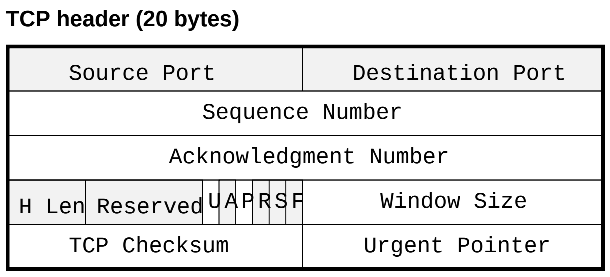

Figure 7: TCP header. Grey fields usually do not change.

Most fields in the TCP header are transmitted as the difference from the previous header. The changes are usually by small positive numbers and the difference can be represented using fewer bits than the absolute value. Differences of 1-255 are represented by one byte and differences of 0 or 256-65535 are represented by three bytes.

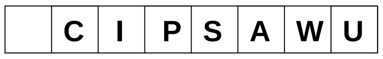

Figure 8: Flag byte of compressed TCP header.

A flag byte, see figure 8, encodes the fields that have changed. Thus no values need to be transmitted for fields that do not change. The S, A, and W bits of the flag byte corresponds to the Sequence Number, Acknowledgment Number, and Window Size fields of the TCP header. The I bit is associated with an identification field in the IPv4 header, encoded in the same way as the previously mentioned fields. The U and P bits in the flag byte are copies of the U and P flags in the TCP header. The Urgent Pointer field is transmitted only when the U bit is set. Finally, the C bit allows the 8-bit CID to be compressed away when several consecutive packets belong to the same TCP connection. If the C bit is zero, the CID is the same as on the previous packet. The TCP checksum is transmitted unmodified.

VJ header compression recognizes two special cases that are very common for the data stream of bulk data transfers and interactive remote login sessions, respectively. Using special encodings of the flag byte, the resulting compressed header is then four bytes, one byte for the flag byte, one byte of the CID, and the two byte TCP checksum.

4.2 Updating TCP compression state

VJ header compression uses a differential encoding technique called *delta encoding* which means that differences in the fields are sent rather than the fields themselves. Using delta encoding implies that the compression state stored in the decompressor changes for each header. When a header is lost, the compression state of the decompressor is not incremented properly and the compressor and decompressor will have inconsistent state. This is different from UDP where loss of compressed headers do not make the state inconsistent. Inconsistent compression state for TCP/IP streams will result in a situation where sequence numbers and/or acknowledgment numbers of decompressed headers are off by some number k, typically the size of the missing segment. The TCP receiver (sender) will compute the TCP checksum which reliably detects such errors and the segment (acknowledgment) will be discarded by the TCP receiver (sender).

TCP receivers do not send acknowledgments for discarded segments, and TCP senders do not use discarded acknowledgments, so the TCP sender will eventually get a timeout signal and retransmit. The compressor peeks into TCP segments and acknowledgments and detects when TCP retransmits, and then sends a full header. The full header updates the compression state at the decompressor and subsequent headers are decompressed correctly.

4.3 Simulated scenarios

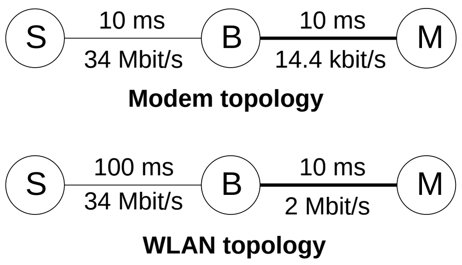

Figure 9: Modem and Wireless LAN (WLAN) topologies. S: Stationary computer, B: Base station or modem server, M: Mobile. Header compression over the bottleneck link (if done). TCP connections between S and M. Right link is lossy.

To investigate the effects of header compression in various scenarios we have used the LBNL Network Simulator [20], a network simulator based on the REAL simulator [21]. A number of TCP variants are available, including TCPs that support selective acknowledgments, and it is possible to set up various network topologies. We have extended the simulator to allow emulation of VJ header compression, and in this paper we show simulations over the two topologies in figure 9. The Modem topology is meant to mirror a path including a low-delay wireless link with 14.4 kbit/s capacity. It represents a path including a GSM or CDPD link. The WLAN topology is meant to mirror a long distance path where the first (or last) hop is over a 2 Mbit/s wireless local area network.

In our simulations, the probability that a transmitted bit is damaged is uniform and independent. This implies that the times between bit-errors are exponentially distributed.

4.4 VJ header compression over low-bandwidth links

VJ header compression works well over connections where the delay-bandwidth product is small, and consequently the sending window is small, as evident from figures 10 and 11. The figures show throughput over the Modem topology. TCP segments have a payload of 512 bytes and have an extra IPv6 header for tunnelling IP datagrams from a *Home Agent* to a mobile host as described in the current Mobile IPv6 draft [15], resulting in a total header of 100 bytes. Compressed headers are assumed to be 5 bytes on average, a slightly pessimistic value for data transfers.

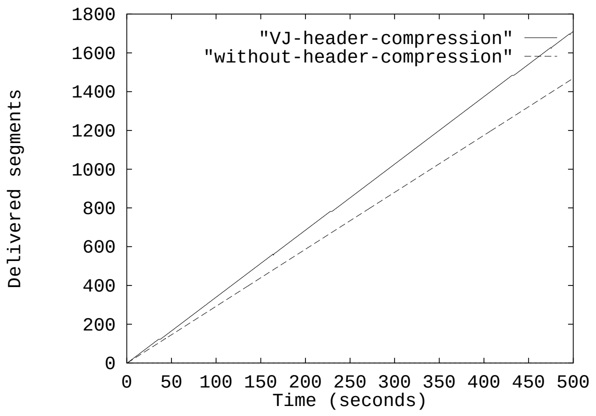

Figure 10: Number of 512 byte segments delivered across Modem topology with bit-error rate 2×10^{-7}.

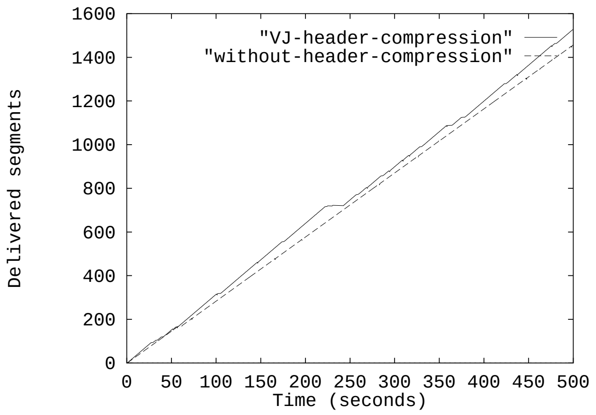

Figure 11: Number of 512 byte segments delivered across Modem topology with bit-error rate 2×10^{-6}.

The curves show performance with and without header compression, for bit-error rates of 2×10^{-7} (figure 10) and 2×10^{-6} (figure 11). With the lower bit-error rate, header compression provides higher throughput corresponding to the reduced packet size, about 16%. With higher bit-error rates, throughput is better with header compression than without. VJ header compression was developed to be used over low-speed links, and even with relatively high bit-error rates, it performs well over such links.

In Figure 10 the curve for header compression has several dips, with big dips around 230 and 360 seconds. These are the result of packet losses. With every loss, acknowledgments stop coming back and the TCP sender will take a timeout before the retransmit which repairs the compression state. There is no similar dip in the curve for no header compression. This is because TCP's *fast retransmit* algorithm is usually able to repair a single lost segment without having to wait for a timeout signal. The fast retransmit algorithm occurs when the TCP sender deduces from a small number of duplicate acknowledgments (usually three) that a segment has been lost, and so retransmits the missing segment. Which segment is missing can be deduced from the duplicate acknowledgments.

Fast retransmit does not work with VJ header compression. A lost data segment causes mismatching compression state between compressor and decompressor, and subsequent data segments will be discarded by the TCP receiver. No acknowledgments will be sent until a retransmission updates the compression state.

In Figure 11, the curve for header compression has a large dip at 230 seconds. This is because the congestion control mechanisms of TCP are triggered by repeated losses and TCP reduces its sending rate. Without header compression, fast retransmit is able to repair lost segments and there are no noticeable dips.

4.5 VJ header compression over medium-bandwidth links

With the coming of IPv6 and Mobile IP there is a need to conserve bandwidth even over medium-speed links, with bit-rates of a few Mbit/s. Moreover, many TCP connections will be across large geographic distances, for example between Europe and USA, and these paths can have significant delays due to propagation, queueing, and processing delays in routers. Figure 12 shows the effects of VJ header compression on a bulk transfer in the WLAN scenario with a moderate bit-error rate on the wireless link. The throughput with header compression drops significantly, from 620 kbit/s to 470 kbit/s or about 25%.

One reason for the reduced throughput is that the delay–bandwidth product is much larger in this scenario. The sending window needs to be at least 50 kbytes to fill the link. With header compression, every

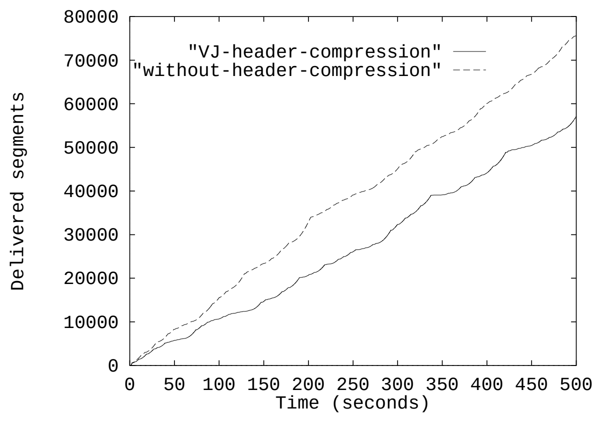

Figure 12: Delivered 512 byte segments across WLAN topology with bit-error rate 2×10^{-7}.

lost segment results in losing a timeout interval's worth of segments due to inconsistent compression state. A timeout has to occur before retransmission and update of the compression state, and the timeout interval is at least equivalent to a round-trip's worth of data, i.e., at least 50 kbyte. With high bit-error rates, this effect alone can severely reduce throughput.

Bit-error rate	10^{-8}	10^{-7}	10^{-6}
Without Hdr Comp			
Segments btw loss (avg)	20400	2040	204
Loss rate	0.0049%	0.049%	0.49%
With Hdr Comp			
Segments btw loss (avg)	24200	2420	242
Loss rate (incl window)	0.40%	4.0%	40%

Figure 13: Effects of Header Compression on loss rate.

The table in figure 13 show some calculations of the effects of packet loss in the WLAN topology when the sending window is assumed to be constant at 50 kbytes. The segment size is 512 bytes and header compression is assumed to reduce the header to 5 bytes. The 50 kbyte window is equivalent to 98 segments. Without header compression, the fast repair mechanism is assumed to be able to repair a loss without triggering a timeout. With header compression, the timeout period is assumed to be exactly equivalent to the round-trip time of 220 ms, which is very optimistic.

Another reason for the reduced throughput of figure 12 is the congestion control mechanisms of TCP. TCP assumes that every lost segment is due to congestion and reduces its sending window for each loss. The sending window determines the amount of data that can be transmitted per round-trip time, so this reduces TCP's sending rate. When the congestion signal is a retransmission timeout, the window is reduced more than what it would be after a fast retransmit. Since header compression disables fast retransmit, the window after a loss will be smaller with header compression than without.

It is clear that repeated loss of whole sending windows combined with additional backoff from the congestion control mechanisms of TCP can result in bad performance over lossy links when traditional header compression is being used.

4.6 Ideal, lossless TCP header compression

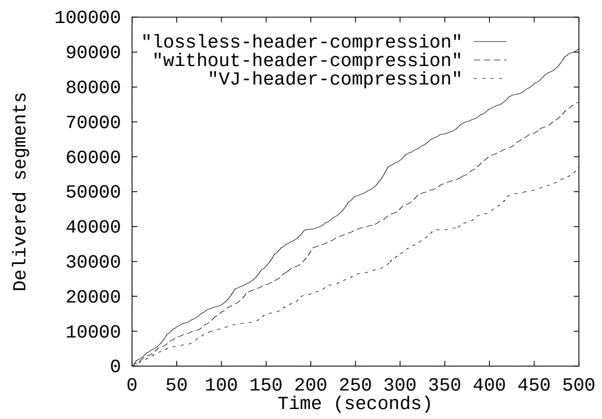

Figure 14: Delivered 512 byte segments over WLAN topology with bit-error rate 2×10^{-7}.

We saw in section 3.5 that the packet loss rate is reduced when headers are smaller. This means that header compression can result in higher throughput because TCP's sending window can grow larger between losses. If the compression state can be repaired quickly, header compression will increase throughput for TCP transfers, as illustrated in Figure 14. The figure plots number of delivered segments for two TCP transfers where the better one experiences 18% less packet loss due to a reduction in header size from 100 bytes to 5 bytes. The increase in throughput is about 28%. Thus, lossless header compression, i.e., header compression where no extra packet loss occurs due to header compression, increases TCP throughput over lossy links significantly.

4.7 Low-loss TCP header compression and the *twice* algorithm

TCP header compression reduces throughput over lossy links because the compression state is not updated properly when packets are lost. This disables acknowledgments, and bandwidth is wasted when segments that were unharmed are retransmitted after a timeout. In this section we describe mechanisms that speed up updating of the compression state. Achieving totally lossless header compression may not be feasible. However, we will show that two simple mechanisms achieve low-loss header compression with comparable performance for bulk data transfers.

A decompressor can detect when its compression state is inconsistent by using the TCP checksum. If it fails, the compression state is deemed inconsistent. A repair can then be attempted by making an educated guess on the properties of the loss. The decompressor assumes that the inconsistency is due to a single lost segment. It then attempts to decompress the received compressed header *again* on the assumption that the lost segment would have incremented the compression state in the same way as the current segment. In this manner the delta of the current segment is applied *twice* to the compression state. If the checksum succeeds, the segment is delivered to IP and the compression state is consistent again.

Figure 16 shows success rates for this simple mechanism, called the *twice* algorithm. The rates were obtained by analyzing packet traces from FTP sessions downloading a 10 Mbyte file to a machine at Luleå University. The *Long* trace is from an ftp site at MIT, the *Medium* trace from a site in Finland, and the *Short* and *LAN* traces from a local ftp site, and a machine on the same Ethernet, respectively. Figure 15 lists information about the traces.

Trace	*RTT (ms)*	*#hops*	*transfer time*
Long	125 – 200	14	26 min
Medium	27 – 32	6	5 min
Short	5 – 18	2	3 min
LAN	0 – 1	0	25 sec

Figure 15: Trace information.

The traces contain a number of TCP connections, including the control connection of FTP. The data and acknowledgment streams are listed separately. Each segment in the compressed traces was examined and for each segment, it was noted whether the *twice* algorithm would be able to repair the compression state if that segment was lost.

The *twice* algorithm performs very well for data streams, with success rates close to 100% for the Medium and Short traces. The Long trace is slightly worse because congestion losses and retransmissions cause varying increments in compressed headers. For the LAN trace, the hard disc was the bottleneck of the transfer. 8192 byte disc blocks were fragmented into five 1460 byte segments, 1460 being the MTU of the Ethernet, and a remaining segment of 892 bytes. This explains the 66.3% success rate for the data segment stream, since the twice algorithm fails 2 times for every

6 segments.

Trace	*Data stream*	*Ack stream*
Long	82.8	45.4
Medium	98.6	97.8
Short	99.3	39.1
LAN	66.3	20.1

Figure 16: Success rates (%) for twice algorithm.

For acknowledgment streams, the success rates are much lower except for the Medium trace. The culprit is the delayed acknowledgement mechanism of TCP where the TCP receiver holds on to an *ack*, usually 200 ms[6], before transmitting it. If additional segments arrive during this time the *ack* will include those too. For the Long and Short traces, 72.0% and 98.8% of all acknowledgments had deltas of one or two times the segment size, respectively. The obvious optimization of the *twice* algorithm, to try multiples of the segment size, would also then reach high success rates for these traces. The combination of varying segment sizes and the delayed *ack* mechanism explains the low success rate for the LAN trace, deltas were usually some low multiple of 1460 plus possibly 892. The most common deltas were 2920 and 3812. The straightforward optimization mentioned above would increase the success rate for the LAN trace to 53%.

When the *twice* algorithm fails to repair the compression state for an acknowledgment stream, a whole window of data will be lost and the TCP sender will receive a timeout signal and do a slow start. Thus, the low success rate for acknowledgment streams call for additional machinery to speed up the repair.

Over a wireless link or LAN, it is highly likely that the two packet streams constituting a TCP connection pass through the same nodes on each side. There will then be a compressor–decompressor pair on each side. A request for sending a full header can thus be passed from decompressor to compressor by setting a flag in the TCP stream going in the opposite direction. This requires communication between compressor and decompressor at both nodes.

When the data segment stream is broken, acknowledgements stop coming back and there is no full header for inserting a header request. So this mechanism will not work for data segment streams. One way to resolve this would be to have the decompressor create and forward a segment containing a single byte that the TCP receiver has already seen. This will cause the TCP receiver to send a duplicate acknowledgment in which the header request can be inserted.

[6]TCP spec allows 500 ms.

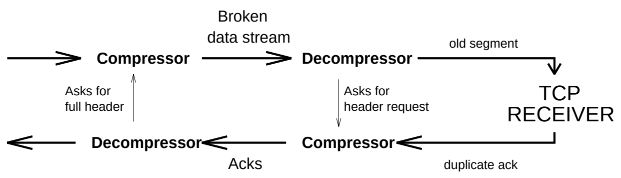

Figure 17: Header request mechanism.

To further improve the situation, the segments received while the data stream is broken could be stored and decompressed later when a retransmission provides the missing segments. Adding these two mechanisms, header compression should be practically lossless. However, the *twice* algorithm performs well on data streams, so it is doubtful whether the extra machinery can be justified. For acknowledgment streams, the request–repair mechanism works well.

Having implemented the *twice* algorithm and the full header request mechanism in the simulator, we ran the ideal lossless header compression algorithm and the low-loss header compression algorithms against each other. Figure 18 shows a typical result. The two header compression curves grow with similar rates, and they are both significantly better than the curve without header compression. Sometimes low-loss header compression is actually *ahead* of the ideal lossless header compression, this is because random effects make them experience slightly different packet losses.

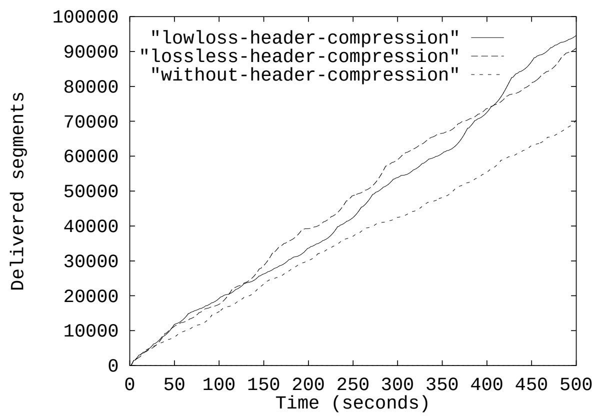

Figure 18: Delivered 512 byte segments over WLAN topology with bit-error rate 2×10^{-7}.

4.8 Performance versus bit-error rate

We ran a series of simulations on the WLAN topology where the bit-error rate varied from 10^{-4} (on average, one segment in 2 is lost) to 10^{-9} (on average, one segment in 200 000 is lost). Figure 19 shows the results for various header compression algorithms. For reference, the performance without header compression is also shown.

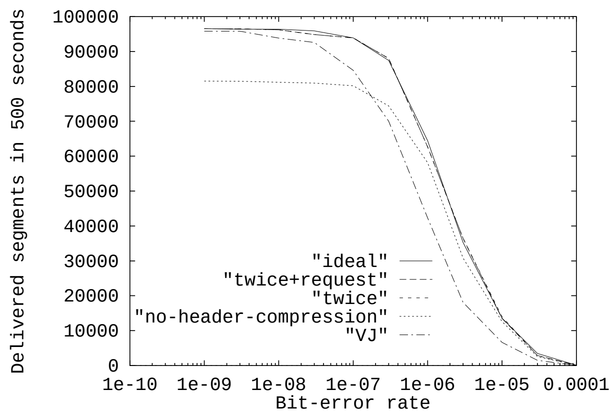

Figure 19: Delivered 512 byte segments over WLAN topology in 500 seconds for different header compression algorithms and different bit-error rates.

We see that the low-loss header compression algorithms perform well for all bit-error rates. They beat VJ header compression when the bit-error rate is low, and are better than no header compression when the bit-error rate is high. In particular, for bit-error rates around 2×10^{-7}, low-loss header compression performs significantly better than both of VJ header compression and no header compression.

The curves for the *twice* algorithm and the *twice* algorithm plus header request are so similar that they cannot be distinguished, which implies that the header request mechanism is not needed when the bit-error probability is uniform and independent. Moreover, the curve for ideal lossless header compression is almost indistinguishable from the low-loss curves. This suggests that it is not possible to improve the low-loss mechanisms significantly.

5 Related work and discussion

The first work on header compression by Jacobson resulted in the now familiar VJ header compression method [10], widely used in the Internet today. VJ header compression can compress TCP/IPv4 headers only, UDP headers are not compressed by his method. Most real-time traffic in the Internet today uses UDP, so there is a need for compression mechanisms for UDP.

Mathur et al [12] has defined a header compression method for IPX, that can be adapted to UDP. In their scheme, compressor and decompressor perform a handshake after each full header. Thus, the scheme in [12] cannot be used over simplex links, and the *ack* implosion problem makes it hard to adapt for multicast communication. The cost of our scheme compared to handshake-based schemes is slightly higher in terms of bandwidth, but the ability to use it for multicast and over simplex links justifies this cost.

With the coming of IPv6 and Mobile IP, there is a need to preserve bandwidth over medium-speed lossy links. For bulk transfers, VJ header compression performs badly over such links, and using it actually reduces throughput. Although the link is less utilized and more users can be served when there is less overhead, most users will not accept decreased performance. We have shown that with extra mechanisms for quick repair of compression state, header compression can increase TCP throughput significantly over lossy links. This is largely due to the reduced packet loss rate that allows TCP to increase its sending window more between losses.

A number of researchers have worked with increasing TCP throughput over lossy wireless links. One example is the Berkeley *snoop protocol* [2], which augments TCP by inserting a booster protocol [9] over the wireless link. The booster protocol stores segments temporarily, snoops into segments and acknowledgments to detect what segments are lost, and performs local retransmissions over the wireless link. This helps increase the performance of TCP because the congestion control mechanisms of TCP are less likely to be triggered and the sending window can open up more than with a standard TCP. The performance of such boosters would be severely reduced if traditional VJ header compression was used because there would be no acknowledgments after a loss.

With low-loss header compression, the throughput with booster protocols should increase. The lower packet loss rate is beneficial because fewer segments need to be retransmitted, and if the booster manages to fill the link to capacity, the reduced header size promises a performance increase of around 15% for IPv6 and Mobile IP headers. Moreover, booster protocols such as in [2] can benefit from the decompressor's detailed knowledge of when packet losses has occurred. It would make sense to have the decompressor inform the booster protocol of when losses occur, and have the booster tell the compressor when to send a full header.

The *twice* algorithm seemed to perform badly for the LAN trace, with success rates of 66% for the data stream and 20% for the acknowledgment stream. The bottleneck, however, was the disc. TCP ran out of data and had to send a smaller segment at the end of each disc block. It is unlikely that this situation will occur on a medium-speed wireless LAN, where the bottleneck of a data transfer is more likely to be in the network than the hard disc.

We have used uniformly distributed bit-error frequencies in our simulations. This implies that most packet losses are for single packets. It is not clear that this is a good model for a wireless LAN. Two recent studies of the AT&T WaveLAN [7, 11] have come to slightly different conclusions. [11] found that withing a room, packet losses does not occur in groups and are uniformly distributed. For longer distances between rooms packet loss occur in groups of 2-3 packets[7]. The other study [7] also found that within a room, losses are uniform and for single packets. This was also true between rooms. Moreover, this latter study found a much lower correlation between distance and loss rate than the previous study.

If it is true that most packet losses occur in groups of one to three packets, the *twice* mechanism should be extended to be able to repair one to three lost packets. The compressor can keep track of the consecutive changes to the TCP header and send an occasional full header to ensure that the TCP checksum will detect all inconsistent decompression resulting from such loss.

If packet losses occur in long groups, the *twice* algorithm will fail and the compression state is not repaired. However, the header request mechanism and sending an empty data segment to ensure that the TCP receiver sends an acknowledgment should improve the situation considerably. Temporary storing data segments that cannot be decompressed for later decompression may or may not be justified, this is a topic for further study.

6 Implementation status

We have a prototype implementation where UDP is used as the link layer. A modified `tcpdump` allows us to capture real packet traces and feed them into the prototype for compression and decompression. Processing times for the prototype are listed in figure 20. Times were measured using `gettimeofday()` on a SUN Sparc-5. Little time has been spent optimizing the code of the prototype, it is likely that the reported times can be improved.

Header	*Compressor*		*Decompressor*	
	avg	*extremes*	*avg*	*extremes*
regular	11	10, 12	7	6, 8
full	31	20, 43	16	15, 17
compr	35	32, 49	27	24, 29

Figure 20: Processing times, microseconds.

The reason for the large variation in the processing times for compression is that the compressor must find the appropriate compression state before compressing. The implementation performs a linear search over the compression state of active CIDs, and the processing time includes this linear search.

[7]In this study, 1400 byte packets were used.

Header compression processing time is low compared to header transmission time. For example, on a 2 Mbit/s link it takes 0.5 μs to transmit one bit. Total processing time for a compressed header is $35 + 27 = 62$ μs, which is equivalent to 15.5 bytes. Since a TCP/IPv6 header is reduced by about 55 bytes with header compression, compressed segments will be delivered sooner with header compression than without.

We are currently implementing IPv6 header compression in the NetBSD kernel, and are planning a Streams module for Sun Microsystems, Inc., Solaris operating system. A current Internet Draft [8] specifies the details of IPv6 header compression.

7 Conclusion

The large headers of IPv6 and Mobile IP threaten to reduce the applicability of Internet technology over low- and medium-speed links. Some delay sensitive applications need to use small packets, for instance remote login and real-time audio applications, and the overhead of large headers on small packets can be prohibitive.

A natural way to alleviate the problem is to compress headers. We have shown how to compress UDP/IP headers, resulting in improved bandwidth efficiency and reduced packet loss rates over lossy wireless links. Our method, based on soft state and periodic header refreshes, can be used over simplex links and for multicast communication. A new mechanism, *Compression Slow-Start*, allows quick installation of compression state and high compression rates.

Since header compression reduces the packet loss rate, using header compression for TCP improves throughput over lossy wireless links. With longer times between packet losses, the TCP sending window can open up more because the congestion control mechanisms are not invoked as often. However, the compression state used by the decompressor must be repaired quickly after a loss, and we present two mechanisms for quick repair of compression state. One mechanism extrapolates what the compression state is likely to be after a loss is detected. Analysis of packet traces show that this method is very efficient. The other mechanism requests a header refresh by utilizing the TCP stream going in the opposite direction.

Simulations show that the resulting low-loss header compression method is better than VJ header compression and better than not doing header compression at all, for bit-error rates from 10^{-9} to 10^{-4}. Low-loss header compression is a win, for delay-sensitive applications as well as bulk data transfers.

8 Acknowledgments

We would like to thank Steve Deering for insightful and valuable comments on our specification of IPv6 header compression and Björn Grönvall for commenting on early versions of this paper. Craig Partridge has ob-

served that in special circumstances, specifically when the Window Scale TCP option is used, the TCP checksum can fail to detect incorrect decompression. This has probably prevented a number of sleepless nights trying to figure out what was going wrong. Now we have mechanisms to avoid this problem. Thanks Craig.

References

[1] Mary G. Baker, Xinhua Zhao, Stuart Cheshire, Jonathan Stone: *Supporting Mobility in MosquitoNet.* Proc. 1996 USENIX Technical Conference, San Diego, CA, January 1996.

[2] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, Randy H. Katz: *Improving TCP/IP performance over Wireless Networks.* Proc. MobiCom '95, Berkeley, CA, November 1995, pp. 2–11.

[3] David D. Clark: *The Design Philosophy of the DARPA Internet Protocols.* Proc. SIGCOMM '88, Computer Communication Review Vol. 18, No. 4, August, 1988, pp. 106–114. Also in Computer Communication Review Vol. 25, No. 1, January, 1995, pp. 102–111.

[4] Steve Deering: *Host Extensions for IP Multicasting.* Request For Comment 1112, August, 1989. `ftp://ds.internic.net/rfc/rfc1112.{ps,txt}`

[5] Steve Deering, Robert Hinden: *Internet Protocol, Version 6 (IPv6) Specification.* Request For Comment 1883, December, 1995. `ftp://ds.internic.net/rfc/rfc1883.txt`

[6] Stephen Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, Liming Wei: *An Architecture for Wide-Area Multicast Routing.* Proc. ACM SigComm '94, Computer Communication Review, Vol. 24, No. 4, October, 1994, pp. 126–135.

[7] David Eckhardt, Peter Steenkiste: *Measurement and Analysis of the Error Characteristics of an In-Building Wireless Network.* Proc. ACM SigComm '96, Computer Communication Review, Vol. 26, No. 4, October, 1996, pp. 243–254.

[8] Mikael Degermark, Björn Nordgren, Stephen Pink: *Header Compression for IPv6.* Internet Engineering Task Force, Internet Draft (work in progress), June, 1996. `draft-degermark-ipv6-hc-01.txt`

[9] We owe the concept of a booster protocol to David Feldmeier and Anthony MacAuley.

[10] Van Jacobson: *Compressing TCP/IP Headers for Low-Speed Serial Links.* Request For Comment 1144, February, 1990. `ftp://ds.internic.net/rfc/rfc1144.{ps,txt}`

[11] Giao T. Nguyen, Randy H. Katz, Brian Noble, Mahadev Satyanarayanan: *A Trace-based Approach for Modeling Wireless Channell Behaviour.* To appear, Proc. of the Winter Simulation Conference, December 1996. `http://daedalus.cs.berkeley.edu/publications/wsc96.ps.gz`

[12] A. Mathur, M. Lewis: *Compressing IPX Headers Over WAN Media (CIPX).* Request For Comment 1553, December, 1993. `ftp://ds.internic.net/rfc/rfc1553.txt`

[13] Craig Partridge: *Gigabit networking.* Addison-Wesley, 1993. ISBN 0-201-56333-9.

[14] Charlie Perkins, ed: *IP Mobility Support.* Internet Engineering Task Force, Internet Draft (work in progress), April 22, 1996. `draft-ietf-mobileip-protocol-16.txt`

[15] Charles Perkins, David B. Johnson: *Mobility Support in IPv6.* Internet Engineering Task Force, Internet Draft (work in progress), January 26, 1996. `draft-ietf-mobileip-ipv6-00.txt`

[16] J. L. Romkey: *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP.* Request For Comment 1055, June, 1988.

[17] W. Simpson: *The Point-to-Point Protocol (PPP).* Request For Comment 1661, July, 1994.

[18] W. Simpson: *PPP in HDLC-like Framing.* Request For Comment 1662, July, 1994.

[19] L. Zhang, S. Deering, D. Estrin, S. Shenker, D. Zappala: *RSVP: A New Resource ReSerVation Protocol.* IEEE Network Magazine, pp. 8-18, September, 1993.

[20] Network Research Group, Lawrence Berkelay National Laboratory. *ns — LBNL Network Simulator.* URL `http://www-nrg.ee.lbl.gov/ns/`

[21] S. Keshav, *The REAL Network Simulator.* URL `http://minnie.cs.adfa.oz.au/REAL/`