Network Working Group
Request for Comments: 1144

V. Jacobson[†]
LBL

February 1990

Compressing TCP/IP Headers for Low-Speed Serial Links

Status of this Memo

This RFC is a proposed elective protocol for the Internet community and requests discussion and suggestions for improvement. It describes a method for compressing the headers of TCP/IP datagrams to improve performance over low speed serial links. The motivation, implementation and performance of the method are described. C code for a sample implementation is given for reference. Distribution of this memo is unlimited.

[†] This work was supported in part by the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

Contents

1 Introduction

As increasingly powerful computers find their way into people's homes, there is growing interest in extending Internet connectivity to those computers. Unfortunately, this extension exposes some complex problems in link-level framing, address assignment, routing, authentication and performance. As of this writing there is active work in all these areas. This memo describes a method that has been used to improve TCP/IP performance over low speed (300 to 19,200 bps) serial links.

The compression proposed here is similar in spirit to the *Thinwire-II* protocol described in [5]. However, this protocol compresses more effectively (the average compressed header is 3 bytes compared to 13 in Thinwire-II) and is both efficient and simple to implement (the Unix implementation is 250 lines of C and requires, on the average, 90μs ($\sim$170 instructions) for a 20MHz MC68020 to compress or decompress a packet).

This compression is specific to TCP/IP datagrams.[1] The author investigated compressing UDP/IP datagrams but found that they were too infrequent to be worth the bother and either there was insufficient datagram-to-datagram coherence for good compression (e.g., name server queries) or the higher level protocol headers overwhelmed the cost of the UDP/IP header (e.g., Sun's RPC/NFS). Separately compressing the IP and the TCP portions of the datagram was also investigated but rejected since it increased the average compressed header size by 50% and doubled the compression and decompression code size.

2 The problem

Internet services one might wish to access over a serial IP link from home range from interactive "terminal" type connections (e.g., telnet, rlogin, xterm) to bulk data transfer (e.g., ftp, smtp, nntp). Header compression is motivated by the need for good interactive response. I.e., the *line efficiency* of a protocol is the ratio of the data to header+data in a datagram. If efficient bulk data transfer is the only objective, it is always possible to make the datagram large enough to approach an efficiency of 100%.

Human-factors studies[15] have found that interactive response is perceived as "bad" when low-level feedback (character echo) takes longer than 100 to 200 ms. Protocol headers interact with this threshold three ways:

(1) If the line is too slow, it may be impossible to fit both the headers and data into a 200 ms window: One typed character results in a 41 byte TCP/IP packet being sent and a 41 byte echo being received. The line speed must be at least 4000 bps to handle these 82 bytes in 200 ms.

[1] The tie to TCP is deeper than might be obvious. In addition to the compression "knowing" the format of TCP and IP headers, certain features of TCP have been used to simplify the compression protocol. In particular, TCP's reliable delivery and the byte-stream conversation model have been used to eliminate the need for any kind of error correction dialog in the protocol (see sec. 4).

(2) Even with a line fast enough to handle packetized typing echo (4800 bps or above), there may be an undesirable interaction between bulk data and interactive traffic: For reasonable line efficiency the bulk data packet size needs to be 10 to 20 times the header size. I.e., the line *maximum transmission unit* or *MTU* should be 500 to 1000 bytes for 40 byte TCP/IP headers. Even with type-of-service queuing to give priority to interactive traffic, a telnet packet has to wait for any in-progress bulk data packet to finish. Assuming data transfer in only one direction, that wait averages half the MTU or 500 ms for a 1024 byte MTU at 9600 bps.

(3) Any communication medium has a maximum signalling rate, the Shannon limit. Based on an AT&T study[2], the Shannon limit for a typical dialup phone line is around 22,000 bps. Since a full duplex, 9600 bps modem already runs at 80% of the limit, modem manufacturers are starting to offer asymmetric allocation schemes to increase effective bandwidth: Since a line rarely has equivalent amounts of data flowing both directions simultaneously, it is possible to give one end of the line more than 11,000 bps by either time-division multiplexing a half-duplex line (e.g., the Telebit Trailblazer) or offering a low-speed "reverse channel" (e.g., the USR Courier HST).[2] In either case, the modem dynamically tries to guess which end of the conversation needs high bandwidth by assuming one end of the conversation is a human (i.e., demand is limited to <300 bps by typing speed). The factor-of-forty bandwidth multiplication due to protocol headers will fool this allocation heuristic and cause these modems to "thrash".

From the above, it's clear that one design goal of the compression should be to limit the bandwidth demand of typing and ack traffic to at most 300 bps. A typical maximum typing speed is around five characters per second[3] which leaves a budget $30 - 5 = 25$ characters for headers or five bytes of header per character typed.[4] Five byte headers solve problems (1) and (3) directly and, indirectly, problem (2): A packet size of 100–200 bytes will easily amortize the cost of a five byte header and offer a user 95–98% of the line bandwidth for

[2] See the excellent discussion of two-wire dialup line capacity in [1], chap. 11. In particular, there is widespread misunderstanding of the capabilities of 'echo-cancelling' modems (such as those conforming to CCITT V.32): Echo-cancellation can offer each side of a two-wire line the full line *bandwidth* but, since the far talker's signal adds to the local 'noise', not the full line *capacity*. The 22Kbps Shannon limit is a hard-limit on data rate through a two-wire telephone connection.

[3] See [13]. Typing bursts or multiple character keystrokes such as cursor keys can exceed this average rate by factors of two to four. However the bandwidth demand stays approximately constant since the TCP Nagle algorithm[8] aggregates traffic with a <200ms interarrival time and the improved header-to-data ratio compensates for the increased data.

[4] A similar analysis leads to essentially the same header size limit for bulk data transfer ack packets. Assuming that the MTU has been selected for "unobtrusive" background file transfers (i.e., chosen so the packet time is 200–400 ms — see sec. 5), there can be at most 5 data packets per second in the "high bandwidth" direction. A reasonable TCP implementation will ack at most every other data packet so at 5 bytes per ack the reverse channel bandwidth is $2.5 \times 5 = 12.5$ bytes/sec.

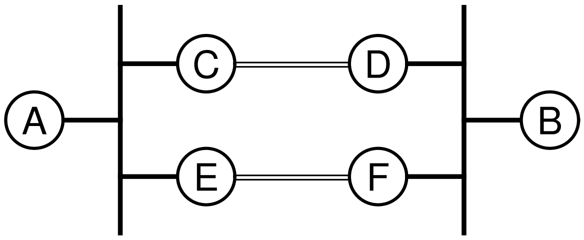

Figure 1: A topology that gives incomplete information at gateways

data. These short packets mean little interference between interactive and bulk data traffic (see sec. 5.2).

Another design goal is that the compression protocol be based solely on information guaranteed to be known to both ends of a single serial link. Consider the topology shown in fig. 1 where communicating hosts A and B are on separate local area nets (the heavy black lines) and the nets are connected by two serial links (the open lines between gateways C–D and E–F).[5] One compression possibility would be to convert each TCP/IP conversation into a semantically equivalent conversation in a protocol with smaller headers, e.g., to an X.25 call. But, because of routing transients or multipathing, it's entirely possible that some of the A–B traffic will follow the A-C-D-B path and some will follow the A-E-F-B path. Similarly, it's possible that A→B traffic will flow A-C-D-B and B→A traffic will flow B-F-E-A. None of the gateways can count on seeing all the packets in a particular TCP conversation and a compression algorithm that works for such a topology cannot be tied to the TCP connection syntax.

A physical link treated as two, independent, simplex links (one each direction) imposes the minimum requirements on topology, routing and pipelining. The ends of each simplex link only have to agree on the most recent packet(s) sent on that link. Thus, although any compression scheme involves shared state, this state is spatially and temporally local and adheres to Dave Clark's principle of *fate sharing*[4]: The two ends can only disagree on the state if the link connecting them is inoperable, in which case the disagreement doesn't matter.

[5] Note that although the TCP endpoints are A and B, in this example compression/decompression must be done at the gateway serial links, i.e., between C and D and between E and F. Since A and B are using IP, they cannot know that their communication path includes a low speed serial link. It is clearly a requirement that compression not break the IP model, i.e., that compression function between intermediate systems and not just between end systems.

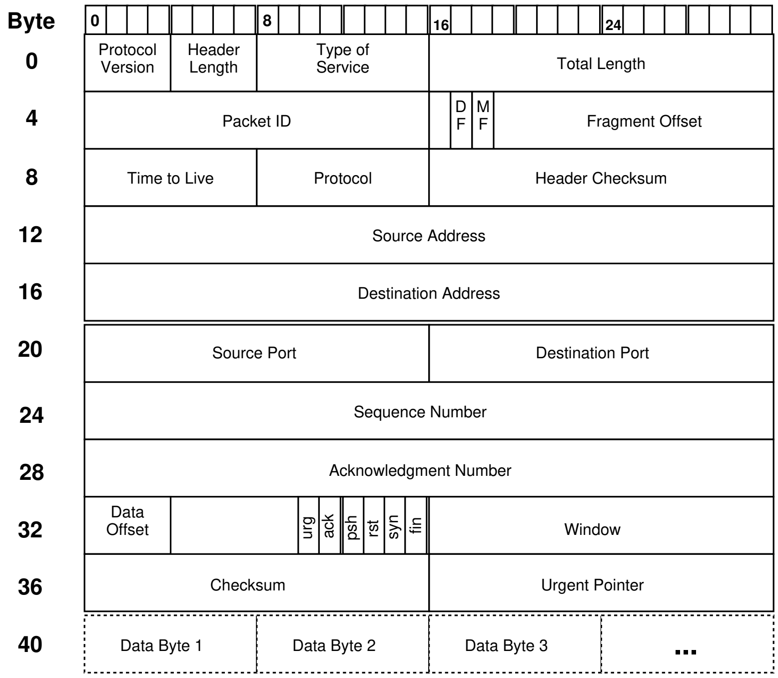

Figure 2: The header of a TCP/IP datagram

3 The compression algorithm

3.1 The basic idea

Figure 2 shows a typical (and minimum length) TCP/IP datagram header.[6] The header size is 40 bytes: 20 bytes of IP and 20 of TCP. Unfortunately, since the TCP and IP protocols were not designed by a committee, all these header fields serve some useful purpose and it's not possible to simply omit some in the name of efficiency.

However, TCP establishes connections and, typically, tens or hundreds of packets are exchanged on each connection. How much of the per-packet information is likely to stay constant over the life of a connection? Half—the shaded fields in fig. 3. So, if the sender and receiver keep track of active connections[7] and the receiver keeps a copy of the header from the last packet it saw from each connection, the sender gets a factor-of-two compression by sending only a small (≤ 8 bit) *connection identifier* together with the 20 bytes that change and letting the receiver fill in the 20 fixed bytes from the saved header.

One can scavenge a few more bytes by noting that any reasonable link-level framing

[6] The TCP and IP protocols and protocol headers are described in [10] and [11].

[7] The 96-bit tuple ⟨*src address, dst address, src port, dst port*⟩ uniquely identifies a TCP connection.

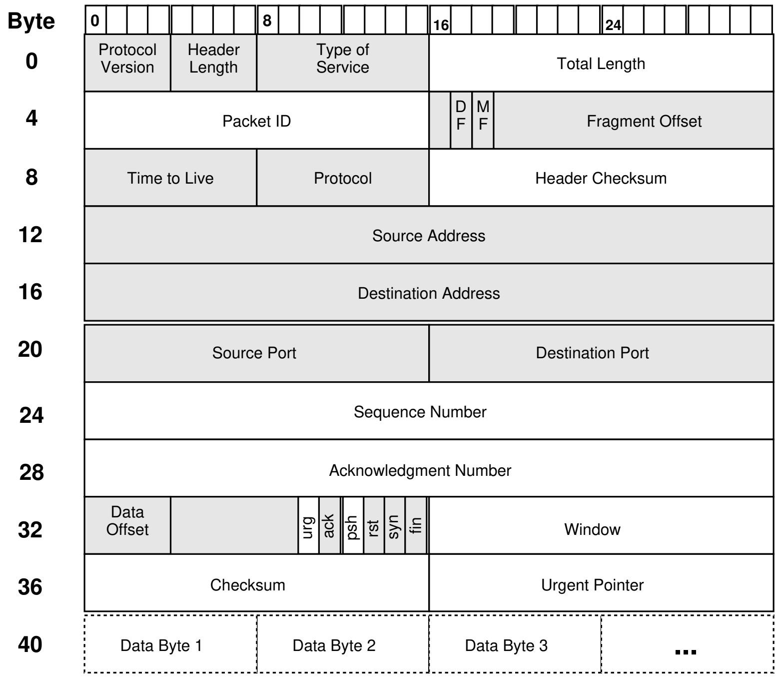

Figure 3: Fields that change during a TCP connection

protocol will tell the receiver the length of a received message so *total length* (bytes 2 and 3) is redundant. But then the *header checksum* (bytes 10 and 11), which protects individual hops from processing a corrupted IP header, is essentially the only part of the IP header being sent. It seems rather silly to protect the transmission of information that isn't being transmitted. So, the receiver can check the header checksum when the header is actually sent (i.e., in an uncompressed datagram) but, for compressed datagrams, regenerate it locally at the same time the rest of the IP header is being regenerated.[8]

This leaves 16 bytes of header information to send. All of these bytes are likely to change over the life of the conversation but they do not all change at the same time. For example, during an FTP data transfer only the *packet ID*, *sequence number* and *checksum* change in the sender→receiver direction and only the *packet ID*, *ack*, *checksum* and, possibly, *window*, change in the receiver→sender direction. With a copy of the last packet sent for each connection, the sender can figure out what fields change in the current packet then

[8] The IP header checksum is *not* an end-to-end checksum in the sense of [14]: The time-to-live update forces the IP checksum to be recomputed at each hop. The author has had unpleasant personal experience with the consequences of violating the *end-to-end argument* in [14] and this protocol is careful to pass the end-to-end TCP checksum through unmodified. See sec. 4.

send a bitmask indicating what changed followed by the changing fields.[9]

If the sender only sends fields that differ, the above scheme gets the average header size down to around ten bytes. However, it's worthwhile looking at how the fields change: The packet ID typically comes from a counter that is incremented by one for each packet sent. I.e., the difference between the current and previous packet IDs should be a small, positive integer, usually <256 (one byte) and frequently $=1$. For packets from the sender side of a data transfer, the sequence number in the current packet will be the sequence number in the previous packet plus the amount of data in the previous packet (assuming the packets are arriving in order). Since IP packets can be at most 64K, the sequence number change must be $<2^{16}$ (two bytes). So, if the *differences* in the changing fields are sent rather than the fields themselves, another three or four bytes per packet can be saved.

That gets us to the five-byte header target. Recognizing a couple of special cases will get us three byte headers for the two most common cases—interactive typing traffic and bulk data transfer—but the basic compression scheme is the differential coding developed above. Given that this intellectual exercise suggests it is possible to get five byte headers, it seems reasonable to flesh out the missing details and actually implement something.

3.2 The ugly details

3.2.1 Overview

Figure 4 shows a block diagram of the compression software. The networking system calls a SLIP output driver with an IP packet to be sent over the serial line. The packet goes through a compressor which checks if the protocol is TCP. Non-TCP packets and "uncompressible" TCP packets (described below) are just marked as TYPE_IP and passed to a framer. Compressible TCP packets are looked up in an array of packet headers. If a matching connection is found, the incoming packet is compressed, the (uncompressed) packet header is copied into the array, and a packet of type COMPRESSED_TCP is sent to the framer. If no match is found, the oldest entry in the array is discarded, the packet header is copied into that slot, and a packet of type UNCOMPRESSED_TCP is sent to the framer. (An UNCOMPRESSED_TCP packet is identical to the original IP packet except the IP protocol field is replaced with a *connection number*—an index into the array of saved, per-connection packet headers. This is how the sender (re-)synchronizes the receiver and "seeds" it with the first, uncompressed packet of a compressed packet sequence.)

The framer is responsible for communicating the packet data, type and boundary (so the decompressor can learn how many bytes came out of the compressor). Since the

[9] This is approximately *Thinwire-I* from [5]. A slight modification is to do a "delta encoding" where the sender subtracts the previous packet from the current packet (treating each packet as an array of 16 bit integers), then sends a 20-bit mask indicating the non-zero differences followed by those differences. If distinct conversations are separated, this is a fairly effective compression scheme (e.g., typically 12-16 byte headers) that doesn't involve the compressor knowing any details of the packet structure. Variations on this theme have been used, successfully, for a number of years (e.g., the Proteon router's serial link protocol[3]).

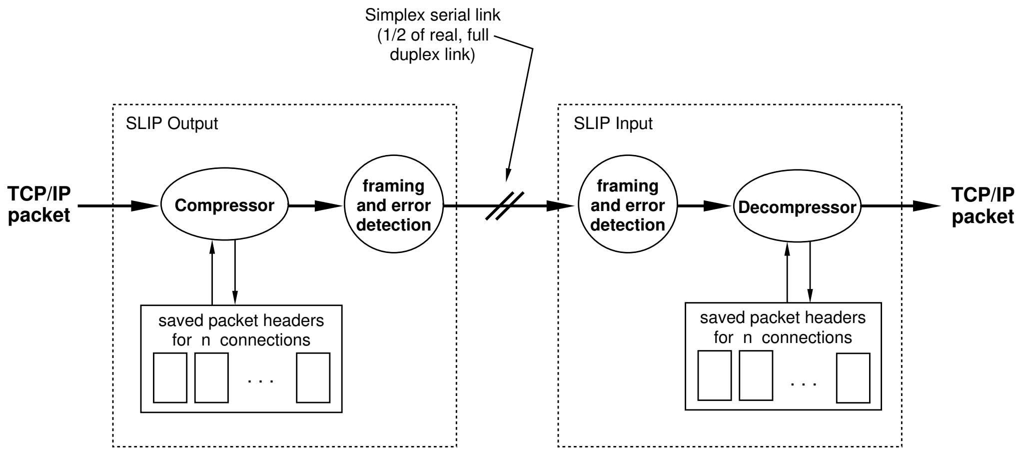

Figure 4: Compression/decompression model

compression is a differential coding, the framer must not re-order packets (this is rarely a concern over a single serial link). It must also provide *good* error detection and, if connection numbers are compressed, must provide an error indication to the decompressor (see sec. 4).[10]

The decompressor does a 'switch' on the type of incoming packets: For TYPE_IP, the packet is simply passed through. For UNCOMPRESSED_TCP, the connection number is extracted from the IP protocol field and IPPROTO_TCP is restored, then the connection number is used as an index into the receiver's array of saved TCP/IP headers and the header of the incoming packet is copied into the indexed slot. For COMPRESSED_TCP, the connection number is used as an array index to get the TCP/IP header of the last packet from that connection, the info in the compressed packet is used to update that header, then a new packet is constructed containing the now-current header from the array concatenated with the data from the compressed packet.

Note that the communication is *simplex*—no information flows in the decompressor-to-compressor direction. In particular, this implies that the decompressor is relying on TCP retransmissions to correct the saved state in the event of line errors (see sec. 4).

3.2.2 Compressed packet format

Figure 5 shows the format of a compressed TCP/IP packet. There is a *change mask* that identifies which of the fields expected to change per-packet actually changed, a *connection number* so the receiver can locate the saved copy of the last packet for this TCP connection,

[10] Link level framing is outside the scope of this document. Any framing that provides the facilities listed in this paragraph should be adequate for the compression protocol. However, the author encourages potential implementors to see [9] for a proposed, standard, SLIP framing.

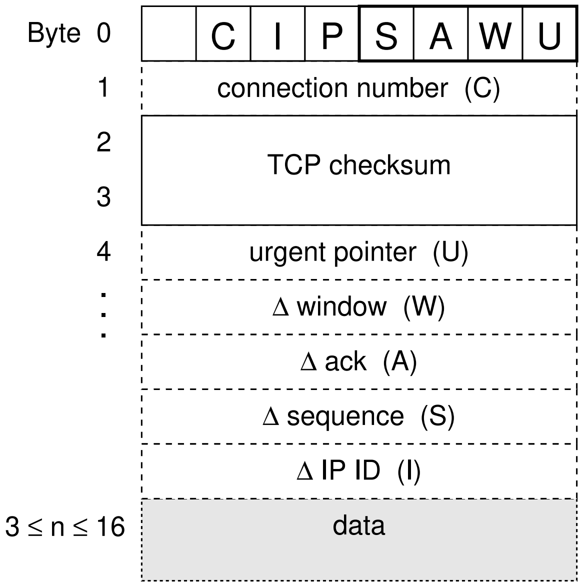

Figure 5: The header of a compressed TCP/IP datagram

the unmodified TCP checksum so the end-to-end data integrity check will still be valid, then for each bit set in the change mask, the amount the associated field changed. (Optional fields, controlled by the mask, are enclosed in dashed lines in the figure.) In all cases, the bit is set if the associated field is present and clear if the field is absent.[11]

Since the delta's in the sequence number, etc., are usually small, particularly if the tuning guidelines in section 5 are followed, all the numbers are encoded in a variable length scheme that, in practice, handles most traffic with eight bits: A change of one through 255 is represented in one byte. Zero is improbable (a change of zero is never sent) so a byte of zero signals an extension: The next two bytes are the MSB and LSB, respectively, of a 16 bit value. Numbers larger than 16 bits force an uncompressed packet to be sent. For example, decimal 15 is encoded as hex `0f`, 255 as `ff`, 65534 as `00 ff fe`, and zero as `00 00 00`. This scheme packs and decodes fairly efficiently: The usual case for both encode and decode executes three instructions on a MC680x0.

The numbers sent for TCP sequence number and ack are the difference[12] between the current value and the value in the previous packet (an uncompressed packet is sent if the difference is negative or more than 64K). The number sent for the window is also the

[11] The bit 'P' in the figure is different from the others: It is a copy of the "PUSH" bit from the TCP header. "PUSH" is a curious anachronism considered indispensable by certain members of the Internet community. Since PUSH can (and does) change in any datagram, an information preserving compression scheme must pass it explicitly.

[12] All differences are computed using two's complement arithmetic.

difference between the current and previous values. However, either positive or negative changes are allowed since the window is a 16 bit field. The packet's urgent pointer is sent if URG is set (an uncompressed packet is sent if the urgent pointer changes but URG is not set). For *packet ID*, the number sent is the difference between the current and previous values. However, unlike the rest of the compressed fields, the assumed change when *I* is clear is one, not zero.

There are two important special cases:

(1) The sequence number and ack both change by the amount of data in the last packet; no window change or URG.

(2) The sequence number changes by the amount of data in the last packet, no ack or window change or URG.

(1) is the case for echoed terminal traffic. (2) is the sender side of non-echoed terminal traffic or a unidirectional data transfer. Certain combinations of the S, A, W and U bits of the change mask are used to signal these special cases. 'U' (urgent data) is rare so two unlikely combinations are S W U (used for case 1) and S A W U (used for case 2). To avoid ambiguity, an uncompressed packet is sent if the actual changes in a packet are S * W U.

Since the 'active' connection changes rarely (e.g., a user will type for several minutes in a telnet window before changing to a different window), the C bit allows the connection number to be elided. If C is clear, the connection is assumed to be the same as for the last compressed or uncompressed packet. If C is set, the connection number is in the byte immediately following the change mask.[13]

From the above, it's probably obvious that compressed terminal traffic usually looks like (in hex): `0B c c d`, where the *0B* indicates case (1), *c c* is the two byte TCP checksum and *d* is the character typed. Commands to *vi* or *emacs*, or packets in the data transfer direction of an FTP 'put' or 'get' look like `0F c c d` ..., and acks for that FTP look like `04 c c a` where *a* is the amount of data being acked.[14]

3.2.3 Compressor processing

The compressor is called with the IP packet to be processed and the compression state structure for the outgoing serial line. It returns a packet ready for final framing and the link level 'type' of that packet.

[13] The connection number is limited to one byte, i.e., 256 simultaneously active TCP connections. In almost two years of operation, the author has never seen a case where more than sixteen connection states would be useful (even in one case where the SLIP link was used as a gateway behind a very busy, 64-port terminal multiplexor). Thus this does not seem to be a significant restriction and allows the protocol field in UNCOMPRESSED_TCP packets to be used for the connection number, simplifying the processing of those packets.

[14] It's also obvious that the change mask changes infrequently and could often be elided. In fact, one can do slightly better by saving the last compressed packet (it can be at most 16 bytes so this isn't much additional state) and checking to see if any of it (except the TCP checksum) has changed. If not, send a packet type that means "compressed TCP, same as last time" and a packet containing only the checksum and data. But, since the improvement is at most 25%, the added complexity and state doesn't seem justified. See appendix C.

As the last section noted, the compressor converts every input packet into either a TYPE_IP, UNCOMPRESSED_TCP or COMPRESSED_TCP packet. A TYPE_IP packet is an unmodified copy[15] of the input packet and processing it doesn't change the compressor's state in any way.

An UNCOMPRESSED_TCP packet is identical to the input packet except the *IP protocol* field (byte 9) is changed from '6' (protocol TCP) to a *connection number*. In addition, the state slot associated with the connection number is updated with a copy of the input packet's IP and TCP headers and the connection number is recorded as the *last connection sent* on this serial line (for the C compression described below).

A COMPRESSED_TCP packet contains the data, if any, from the original packet but the IP and TCP headers are completely replaced with a new, compressed header. The connection state slot and *last connection sent* are updated by the input packet exactly as for an UNCOMPRESSED_TCP packet.

The compressor's decision procedure is:

- If the packet is not protocol TCP, send it as TYPE_IP.
- If the packet is an IP fragment (i.e., either the *fragment offset* field is non-zero or the *more fragments* bit is set), send it as TYPE_IP.[16]
- If any of the TCP control bits *SYN, FIN* or *RST* are set or if the *ACK* bit is clear, consider the packet uncompressible and send it as TYPE_IP.[17]

If a packet makes it through the above checks, it will be sent as either UNCOMPRESSED_TCP or COMPRESSED_TCP:

- If no connection state can be found that matches the packet's source and destination IP addresses and TCP ports, some state is reclaimed (which should probably be the least recently used) and an UNCOMPRESSED_TCP packet is sent.

---

[15] It is not necessary (or desirable) to actually duplicate the input packet for any of the three output types. Note that the compressor cannot increase the size of a datagram. As the code in appendix A shows, the protocol can be implemented so all header modifications are made 'in place'.

[16] Only the first fragment contains the TCP header so the fragment offset check is necessary. The first fragment might contain a complete TCP header and, thus, could be compressed. However the check for a complete TCP header adds quite a lot of code and, given the arguments in [6], it seems reasonable to send all IP fragments uncompressed.

[17] The ACK test is redundant since a standard conforming implementation must set ACK in all packets except for the initial SYN packet. However, the test costs nothing and avoids turning a bogus packet into a valid one.

SYN packets are not compressed because only half of them contain a valid ACK field and they usually contain a TCP option (the max. segment size) which the following packets don't. Thus the next packet would be sent uncompressed because the TCP header length changed and sending the SYN as UNCOMPRESSED_TCP instead of TYPE_IP would buy nothing.

The decision to not compress FIN packets is questionable. Discounting the trick in appendix B.1, there is a free bit in the header that could be used to communicate the FIN flag. However, since connections tend to last for many packets, it seemed unreasonable to dedicate an entire bit to a flag that would only appear once in the lifetime of the connection.

- If a connection state is found, the packet header it contains is checked against the current packet to make sure there were no unexpected changes. (E.g., that all the shaded fields in fig. 3 are the same). The IP protocol, fragment offset, more fragments, SYN, FIN and RST fields were checked above and the source and destination address and ports were checked as part of locating the state. So the remaining fields to check are *protocol version, header length, type of service, don't fragment, time-to-live, data offset*, IP options (if any) and TCP options (if any). If any of these fields differ between the two headers, an UNCOMPRESSED_TCP packet is sent.

If all the "unchanging" fields match, an attempt is made to compress the current packet:

- If the *URG* flag is set, the *urgent data* field is encoded (note that it may be zero) and the U bit is set in the change mask. Unfortunately, if URG is clear, the urgent data field must be checked against the previous packet and, if it changes, an UNCOMPRESSED_TCP packet is sent. ('Urgent data' shouldn't change when URG is clear but [11] doesn't require this.)

- The difference between the current and previous packet's *window* field is computed and, if non-zero, is encoded and the W bit is set in the change mask.

- The difference between *ack* fields is computed. If the result is less than zero or greater than $2^{16} - 1$, an UNCOMPRESSED_TCP packet is sent.[18] Otherwise, if the result is non-zero, it is encoded and the A bit is set in the change mask.

- The difference between *sequence number* fields is computed. If the result is less than zero or greater than $2^{16} - 1$, an UNCOMPRESSED_TCP packet is sent.[19] Otherwise, if the result is non-zero, it is encoded and the S bit is set in the change mask.

Once the U, W, A and S changes have been determined, the special-case encodings can be checked:

- If *U, S* and *W* are set, the changes match one of the special-case encodings. Send an UNCOMPRESSED_TCP packet.

- If only *S* is set, check if the change equals the amount of user data in the last packet. I.e., subtract the TCP and IP header lengths from the last packet's *total length* field and compare the result to the S change. If they're the same, set the change mask to SAWU (the special case for "unidirectional data transfer") and discard the encoded sequence number change (the decompressor can reconstruct it since it knows the last packet's total length and header length).

[18] The two tests can be combined into a single test of the most significant 16 bits of the difference being non-zero.

[19] A negative sequence number change probably indicates a retransmission. Since this may be due to the decompressor having dropped a packet, an uncompressed packet is sent to re-sync the decompressor (see sec. 4).

- If only *S* and *A* are set, check if they both changed by the same amount and that amount is the amount of user data in the last packet. If so, set the change mask to SWU (the special case for "echoed interactive" traffic) and discard the encoded changes.
- If nothing changed, check if this packet has no user data (in which case it is probably a duplicate ack or window probe) or if the previous packet contained user data (which means this packet is a retransmission on a connection with no pipelining). In either of these cases, send an UNCOMPRESSED_TCP packet.

Finally, the TCP/IP header on the outgoing packet is replaced with a compressed header:

- The change in the *packet ID* is computed and, if not one,[20] the difference is encoded (note that it may be zero or negative) and the I bit is set in the change mask.
- If the *PUSH* bit is set in the original datagram, the P bit is set in the change mask.
- The TCP and IP headers of the packet are copied to the connection state slot.
- The TCP and IP headers of the packet are discarded and a new header is prepended consisting of (in reverse order):
  - the accumulated, encoded changes.
  - the *TCP checksum* (if the new header is being constructed "in place", the checksum may have been overwritten and will have to be taken from the header copy in the connection state or saved in a temporary before the original header is discarded).
  - the *connection number* (if different than the last one sent on this serial line). This also means that the the line's *last connection sent* must be set to the *connection number* and the *C* bit set in the change mask.
  - the change mask.

At this point, the compressed TCP packet is passed to the framer for transmission.

3.2.4 Decompressor processing

Because of the simplex communication model, processing at the decompressor is much simpler than at the compressor — all the decisions have been made and the decompressor simply does what the compressor has told it to do.

[20] Note that the test here is against *one*, not *zero*. The packet ID is typically incremented by one for each packet sent so a change of zero is very unlikely. A change of one is likely: It occurs during any period when the originating system has activity on only one connection.

The decompressor is called with the incoming packet,[21] the length and type of the packet and the compression state structure for the incoming serial line. A (possibly re-constructed) IP packet will be returned.

The decompressor can receive four types of packet: the three generated by the compressor and a TYPE_ERROR pseudo-packet generated when the receive framer detects an error.[22] The first step is a 'switch' on the packet type:

- If the packet is TYPE_ERROR or an unrecognized type, a 'toss' flag is set in the state to force COMPRESSED_TCP packets to be discarded until one with the C bit set or an UNCOMPRESSED_TCP packet arrives. Nothing (a null packet) is returned.

- If the packet is TYPE_IP, an unmodified copy of it is returned and the state is not modified.

- If the packet is UNCOMPRESSED_TCP, the state index from the IP protocol field is checked.[23] If it's illegal, the toss flag is set and nothing is returned. Otherwise, the toss flag is cleared, the index is copied to the state's *last connection received* field, a copy of the input packet is made,[24] the TCP protocol number is restored to the IP protocol field, the packet header is copied to the indicated state slot, then the packet copy is returned.

If the packet was not handled above, it is COMPRESSED_TCP and a new TCP/IP header has to be synthesized from information in the packet plus the last packet's header in the state slot. First, the explicit or implicit connection number is used to locate the state slot:

- If the C bit is set in the change mask, the state index is checked. If it's illegal, the toss flag is set and nothing is returned. Otherwise, *last connection received* is set to the packet's state index and the toss flag is cleared.

- If the C bit is clear and the toss flag is set, the packet is ignored and nothing is returned.

At this point, *last connection received* is the index of the appropriate state slot and the first byte(s) of the compressed packet (the change mask and, possibly, connection index)

[21] It's assumed that link-level framing has been removed by this point and the packet and length do *not* include type or framing bytes.

[22] No data need be associated with a TYPE_ERROR packet. It exists so the receive framer can tell the decompressor that there may be a gap in the data stream. The decompressor uses this as a signal that packets should be tossed until one arrives with an explicit connection number (C bit set). See the last part of sec. 4.1 for a discussion of why this is necessary.

[23] State indices follow the C language convention and run from 0 to $N - 1$, where $0 < N \leq 256$ is the number of available state slots.

[24] As with the compressor, the code can be structured so no copies are done and all modifications are done in-place. However, since the output packet can be larger than the input packet, 128 bytes of free space must be left at the front of the input packet buffer to allow room to prepend the TCP/IP header.

have been consumed. Since the TCP/IP header in the state slot must end up reflecting the newly arrived packet, it's simplest to apply the changes from the packet to that header then construct the output packet from that header concatenated with the data from the input packet. (In the following description, 'saved header' is used as an abbreviation for 'the TCP/IP header saved in the state slot'.)

- The next two bytes in the incoming packet are the TCP checksum. They are copied to the saved header.
- If the P bit is set in the change mask, the TCP PUSH bit is set in the saved header. Otherwise the PUSH bit is cleared.
- If the low order four bits (S, A, W and U) of the change mask are all set (the 'unidirectional data' special case), the amount of user data in the last packet is calculated by subtracting the TCP and IP header lengths from the IP total length in the saved header. That amount is then added to the TCP sequence number in the saved header.
- If S, W and U are set and A is clear (the 'terminal traffic' special case), the amount of user data in the last packet is calculated and added to both the TCP sequence number and ack fields in the saved header.
- Otherwise, the change mask bits are interpreted individually in the order that the compressor set them:
  - If the U bit is set, the TCP URG bit is set in the saved header and the next byte(s) of the incoming packet are decoded and stuffed into the TCP Urgent Pointer. If the U bit is clear, the TCP URG bit is cleared.
  - If the W bit is set, the next byte(s) of the incoming packet are decoded and added to the TCP window field of the saved header.
  - If the A bit is set, the next byte(s) of the incoming packet are decoded and added to the TCP ack field of the saved header.
  - If the S bit is set, the next byte(s) of the incoming packet are decoded and added to the TCP sequence number field of the saved header.
- If the I bit is set in the change mask, the next byte(s) of the incoming packet are decoded and added to the IP ID field of the saved packet. Otherwise, one is added to the IP ID.

At this point, all the header information from the incoming packet has been consumed and only data remains. The length of the remaining data is added to the length of the saved IP and TCP headers and the result is put into the saved IP total length field. The saved IP header is now up to date so its checksum is recalculated and stored in the IP checksum field. Finally, an output datagram consisting of the saved header concatenated with the remaining incoming data is constructed and returned.

4 Error handling

4.1 Error detection

In the author's experience, dialup connections are particularly prone to data errors. These errors interact with compression in two different ways:

First is the local effect of an error in a compressed packet. All error detection is based on redundancy yet compression has squeezed out almost all the redundancy in the TCP and IP headers. In other words, the decompressor will happily turn random line noise into a perfectly valid TCP/IP packet.[25] One could rely on the TCP checksum to detect corrupted compressed packets but, unfortunately, some rather likely errors will not be detected. For example, the TCP checksum will often not detect two single bit errors separated by 16 bits. For a V.32 modem signalling at 2400 baud with 4 bits/baud, any line hit lasting longer than 400μs. would corrupt 16 bits. According to [2], residential phone line hits of up to 2ms. are likely.

The correct way to deal with this problem is to provide for error detection at the framing level. Since the framing (at least in theory) can be tailored to the characteristics of a particular link, the detection can be as light or heavy-weight as appropriate for that link.[26] Since packet error detection is done at the framing level, the decompressor simply assumes that it will get an indication that the current packet was received with errors. (The decompressor always ignores (discards) a packet with errors. However, the indication is needed to prevent the error being propagated — see below.)

The "discard erroneous packets" policy gives rise to the second interaction of errors and compression. Consider the following conversation:

original	*sent*	*received*	*reconstructed*
1: A	1: A	1: A	1: A
2: BC	$\Delta 1$, BC	$\Delta 1$, BC	2: BC
4: DE	$\Delta 2$, DE	—	—
6: F	$\Delta 2$, F	$\Delta 2$, F	4: F
7: GH	$\Delta 1$, GH	$\Delta 1$, GH	5: GH

(Each entry above has the form "*starting sequence number*:*data sent*" or "Δ*sequence number change*,*data sent*".) The first thing sent is an uncompressed packet, followed by four compressed packets. The third packet picks up an error and is discarded. To reconstruct the fourth packet, the receiver applies the sequence number change from incoming compressed packet to the sequence number of the last correctly received packet, packet two, and generates an incorrect sequence number for packet four. After the error, all reconstructed

[25] modulo the TCP checksum.

[26] While appropriate error detection is link dependent, the CCITT CRC used in [9] strikes an excellent balance between ease of computation and robust error detection for a large variety of links, particularly at the relatively small packet sizes needed for good interactive response. Thus, for the sake of interoperability, the framing in [9] should be used unless there is a truly compelling reason to do otherwise.

packets' sequence numbers will be in error, shifted down by the amount of data in the missing packet.[27]

Without some sort of check, the preceding error would result in the receiver invisibly losing two bytes from the middle of the transfer (since the decompressor regenerates sequence numbers, the packets containing F and GH arrive at the receiver's TCP with exactly the sequence numbers they would have had if the DE packet had never existed). Although some TCP conversations can survive missing data[28] it is not a practice to be encouraged. Fortunately the TCP checksum, since it is a simple sum of the packet contents *including the sequence numbers*, detects 100% of these errors. E.g., the receiver's computed checksum for the last two packets above always differs from the packet checksum by two.

Unfortunately, there is a way for the TCP checksum protection described above to fail if the changes in an incoming compressed packet are applied to the wrong conversation: Consider two active conversations C_1 and C_2 and a packet from C_1 followed by two packets from C_2. Since the connection number doesn't change, it's omitted from the second C_2 packet. But, if the first C_2 packet is received with a CRC error, the second C_2 packet will mistakenly be considered the next packet in C_1. Since the C_2 checksum is a random number with respect to the C_1 sequence numbers, there is at least a 2^{-16} probability that this packet will be accepted by the C_1 TCP receiver.[29] To prevent this, after a CRC error indication from the framer the receiver discards packets until it receives either a COMPRESSED_TCP packet with the C bit set or an UNCOMPRESSED_TCP packet. I.e., packets are discarded until the receiver gets an explicit connection number.

To summarize this section, there are two different types of errors: per-packet corruption and per-conversation loss-of-sync. The first type is detected at the decompressor from a link-level CRC error, the second at the TCP receiver from a (guaranteed) invalid TCP checksum. The combination of these two independent mechanisms ensures that erroneous packets are discarded.

4.2 Error recovery

The previous section noted that after a CRC error the decompressor will introduce TCP checksum errors in every uncompressed packet. Although the checksum errors prevent data stream corruption, the TCP conversation won't be terribly useful until the decompressor again generates valid packets. How can this be forced to happen?

The decompressor generates invalid packets because its state (the saved 'last packet header') disagrees with the compressor's state. An UNCOMPRESSED_TCP packet will correct the decompressor's state. Thus error recovery amounts to forcing an uncompressed packet out of the compressor whenever the decompressor is (or might be) confused.

[27] This is an example of a generic problem with differential or delta encodings known as "losing DC".

[28] Many system managers claim that holes in an NNTP stream are more valuable than the data.

[29] With worst-case traffic, this probability translates to one undetected error every three hours over a 9600 baud line with a 30% error rate).

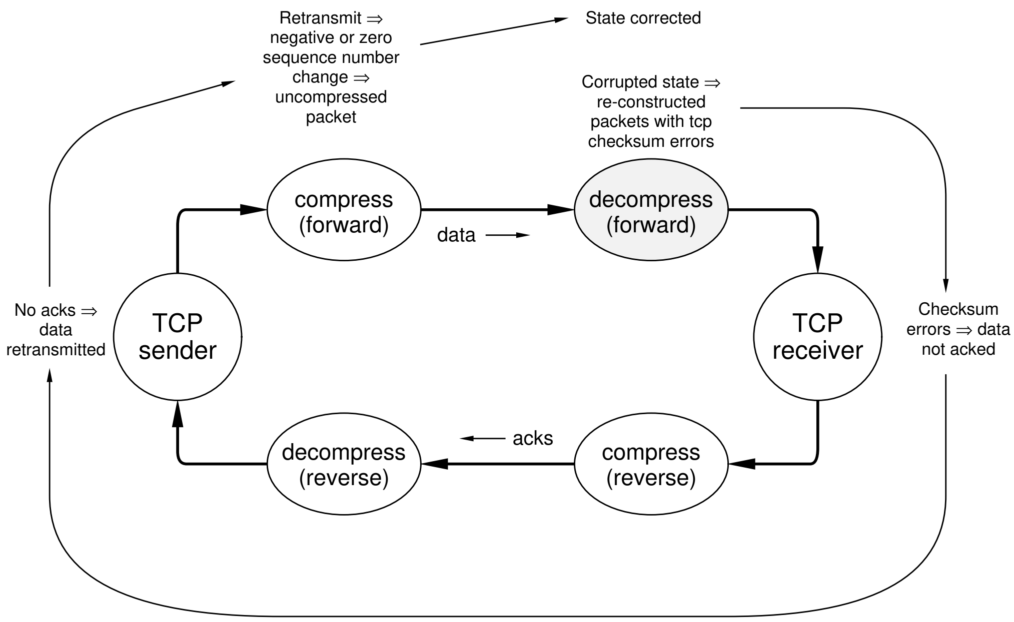

Figure 6: Forward path error correction sequence

The first thought is to take advantage of the full duplex communication link and have the decompressor send something to the compressor requesting an uncompressed packet. This is clearly undesirable since it constrains the topology more than the minimum suggested in sec. 2 and requires that a great deal of protocol be added to both the decompressor and compressor. A little thought convinces one that this alternative is not only undesirable, it simply won't work: Compressed packets are small and it's likely that a line hit will so completely obliterate one that the decompressor will get nothing at all. Thus packets are reconstructed incorrectly (because of the missing compressed packet) but only the TCP end points, not the decompressor, know that the packets are incorrect.

But the TCP end points know about the error and TCP is a reliable protocol designed to run over unreliable media. This means the end points must eventually take some sort of error recovery action and there's an obvious trigger for the compressor to resync the decompressor: send uncompressed packets whenever TCP is doing error recovery.

But how does the compressor recognize TCP error recovery? Consider the schematic TCP data transfer of fig. 6. The confused decompressor is in the forward (data transfer) half of the TCP conversation. The receiving TCP discards packets rather than acking them (because of the checksum errors), the sending TCP eventually times out and retransmits a packet, and the forward path compressor finds that the difference between the sequence number in the retransmitted packet and the sequence number in the last packet seen is either negative (if there were multiple packets in transit) or zero (one packet in transit). The first

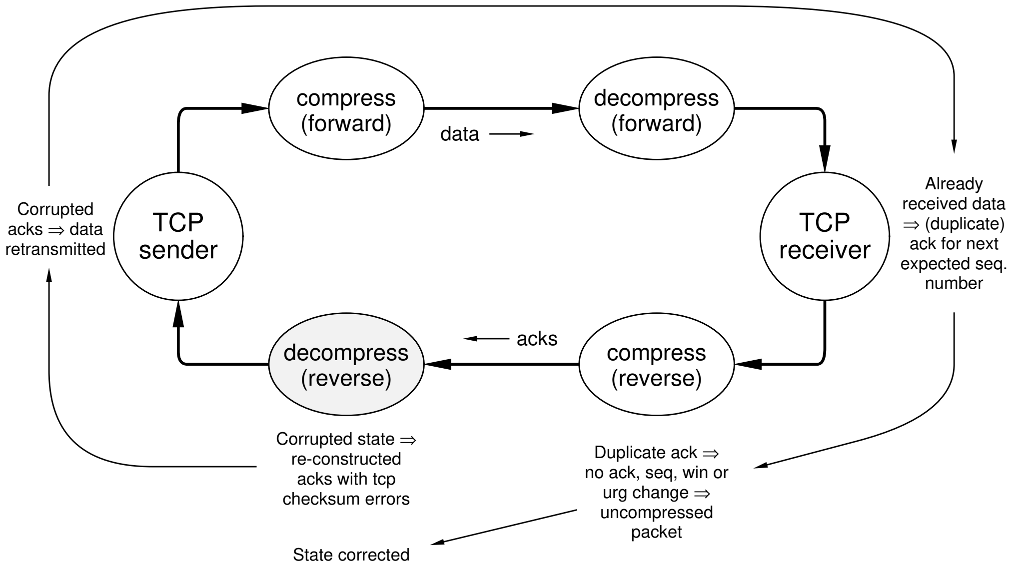

Figure 7: Reverse path error correction sequence

case is detected in the compression step that computes sequence number differences. The second case is detected in the step that checks the 'special case' encodings but needs an additional test: It's fairly common for an interactive conversation to send a dataless ack packet followed by a data packet. The ack and data packet will have the same sequence numbers yet the data packet is not a retransmission. To prevent sending an unnecessary uncompressed packet, the length of the previous packet should be checked and, if it contained data, a zero sequence number change must indicate a retransmission.

A confused decompressor in the reverse (ack) half of the conversation is as easy to detect (fig. 7): The sending TCP discards acks (because they contain checksum errors), eventually times out, then retransmits some packet. The receiving TCP thus gets a duplicate packet and must generate an ack for the next expected sequence number[11, p. 69]. This ack will be a duplicate of the last ack the receiver generated so the reverse-path compressor will find no ack, seq number, window or urg change. If this happens for a packet that contains no data, the compressor assumes it is a duplicate ack sent in response to a retransmit and sends an UNCOMPRESSED_TCP packet.[30]

[30] The packet could be a zero-window probe rather than a retransmitted ack but window probes should be infrequent and it does no harm to send them uncompressed.

5 Configurable parameters and tuning

5.1 Compression configuration

There are two configuration parameters associated with header compression: Whether or not compressed packets should be sent on a particular line and, if so, how many state slots (saved packet headers) to reserve. There is also one link-level configuration parameter, the maximum packet size or MTU, and one front-end configuration parameter, data compression, that interact with header compression. Compression configuration is discussed in this section. MTU and data compression are discussed in the next two sections.

There are some hosts (e.g., low end PCs) which may not have enough processor or memory resources to implement this compression. There are also rare link or application characteristics that make header compression unnecessary or undesirable. And there are many existing SLIP links that do not currently use this style of header compression. For the sake of interoperability, serial line IP drivers that allow header compression should include some sort of user configurable flag to disable compression (see appendix B.2).[31]

If compression is enabled, the compressor must be sure to never send a connection id (state index) that will be dropped by the decompressor. E.g., a black hole is created if the decompressor has sixteen slots and the compressor uses twenty.[32] Also, if the compressor is allowed too few slots, the LRU allocator will thrash and most packets will be sent as UNCOMPRESSED_TCP. Too many slots and memory is wasted.

Experimenting with different sizes over the past year, the author has found that eight slots will thrash (i.e., the performance degradation is noticeable) when many windows on a multi-window workstation are simultaneously in use or the workstation is being used as a gateway for three or more other machines. Sixteen slots were never observed to thrash. (This may simply be because a 9600 bps line split more than 16 ways is already so overloaded that the additional degradation from round-robbining slots is negligible.)

Each slot must be large enough to hold a maximum length TCP/IP header of 128 bytes[33] so 16 slots occupy 2KB of memory. In these days of 4 Mbit RAM chips, 2KB seems so little memory that the author recommends the following configuration rules:

(1) If the framing protocol does not allow negotiation, the compressor and decompressor should provide sixteen slots, zero through fifteen.

[31] The PPP protocol in [9] allows the end points to negotiate compression so there is no interoperability problem. However, there should still be a provision for the system manager at each end to control whether compression is negotiated on or off. And, obviously, compression should default to 'off' until it has been negotiated 'on'.

[32] Strictly speaking, there's no reason why the connection id should be treated as an array index. If the decompressor's states were kept in a hash table or other associative structure, the connection id would be a key, not an index, and performance with too few decompressor slots would only degrade enormously rather than failing altogether. However, an associative structure is substantially more costly in code and cpu time and, given the small per-slot cost (128 bytes of memory), it seems reasonable to design for slot arrays at the decompressor and some (possibly implicit) communication of the array size.

[33] The maximum header length, fixed by the protocol design, is 64 bytes of IP and 64 bytes of TCP.

(2) If the framing protocol allows negotiation, any mutually agreeable number of slots from 1 to 256 should be negotiable.[34] If number of slots is not negotiated, or until it is negotiated, both sides should assume sixteen.

(3) If you have complete control of all the machines at both ends of every link and none of them will ever be used to talk to machines outside of your control, you are free to configure them however you please, ignoring the above. However, when your little eastern-block dictatorship collapses (as they all eventually seem to), be aware that a large, vocal, and not particularly forgiving Internet community will take great delight in pointing out to anyone willing to listen that you have misconfigured your systems and are not interoperable.

5.2 Choosing a maximum transmission unit

From the discussion in sec. 2, it seems desirable to limit the maximum packet size (*MTU*) on any line where there might be interactive traffic and multiple active connections (to maintain good interactive response between the different connections competing for the line). The obvious question is "how much does this hurt throughput?" It doesn't.

Figure 8 shows how user data throughput[35] scales with MTU with (solid line) and without (dashed line) header compression. The dotted lines show what MTU corresponds to a 200 ms packet time at 2400, 9600 and 19,200 bps. Note that with header compression even a 2400 bps line can be responsive yet have reasonable throughput (83%).[36]

Figure 9 shows how line efficiency scales with increasing line speed, assuming that a 200ms. MTU is always chosen.[37] The knee in the performance curve is around 2400 bps. Below this, efficiency is sensitive to small changes in speed (or MTU since the two are linearly related) and good efficiency comes at the expense of good response. Above 2400bps the curve is flat and efficiency is relatively independent of speed or MTU. In other words, it is possible to have both good response and high line efficiency.

To illustrate, note that for a 9600 bps line with header compression there is essentially no benefit in increasing the MTU beyond 200 bytes: If the MTU is increased to 576, the average delay increases by 188% while throughput only improves by 3% (from 96 to 99%).

[34] Allowing only one slot may make the compressor code more complex. Implementations should avoid offering one slot if possible and compressor implementations may disable compression if only one slot is negotiated.

[35] The vertical axis is in percent of line speed. E.g., '95' means that 95% of the line bandwidth is going to *user data* or, in other words, the user would see a data transfer rate of 9120 bps on a 9600 bps line. Four bytes of link-level (framer) encapsulation in addition to the TCP/IP or compressed header were included when calculating the relative throughput. The 200 ms packet times were computed assuming an asynchronous line using 10 bits per character (8 data bits, 1 start, 1 stop, no parity).

[36] However, the 40 byte TCP MSS required for a 2400 bps line might stress-test your TCP implementation.

[37] For a typical async line, a 200ms. MTU is simply $.02\times$ the line speed in bits per second.

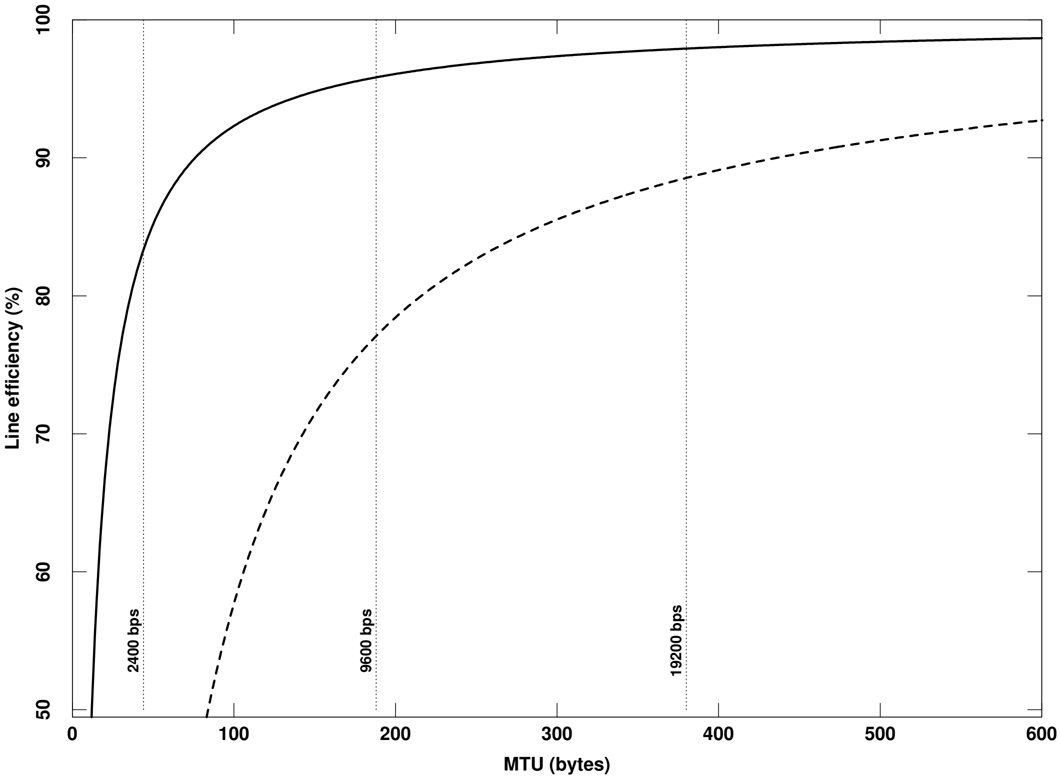

Figure 8: Effective Throughput vs. MTU

5.3 Interaction with data compression

Since the early 1980's, fast, effective, data compression algorithms such as Lempel-Ziv[7] and programs that embody them, such as the *compress* program shipped with Berkeley Unix, have become widely available. When using low speed or long haul lines, it has become common practice to compress data before sending it. For dialup connections, this compression is often done in the modems, independent of the communicating hosts. Some interesting issues would seem to be: (1) Given a good data compressor, is there any need for header compression? (2) Does header compression interact with data compression? (3) Should data be compressed before or after header compression?[38]

To investigate (1), Lempel-Ziv compression was done on a trace of 446 TCP/IP packets taken from the user's side of a typical telnet conversation. Since the packets resulted from typing, almost all contained only one data byte plus 40 bytes of header. I.e., the test essentially measured L-Z compression of TCP/IP headers. The compression ratio (the

[38] The answers, for those who wish to skip the remainder of this section, are 'yes', 'no' and 'either', respectively.

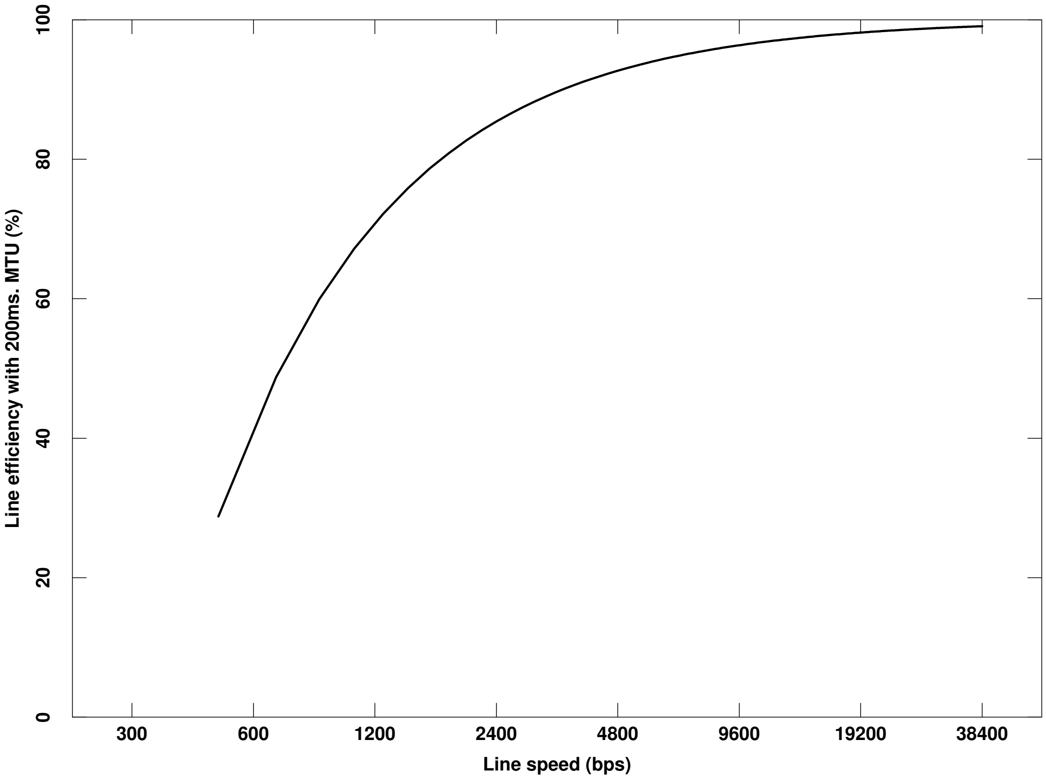

Figure 9: Small MTU line efficiency vs. line speed

ratio of uncompressed to compressed data) was 2.6. In other words, the average header was reduced from 40 to 16 bytes. While this is good compression, it is far from the 5 bytes of header needed for good interactive response and far from the 3 bytes of header (a compression ratio of 13.3) that header compression yielded on the same packet trace.

The second and third questions are more complex. To investigate them, several packet traces from FTP file transfers were analyzed[39] with and without header compression and with and without L-Z compression. The L-Z compression was tried at two places in the outgoing data stream (fig. 10): (1) just before the data was handed to TCP for encapsulation (simulating compression done at the 'application' level) and (2) after the data was encapsulated (simulating compression done in the modem). Table 1 summarizes the results for a 78,776 byte ASCII text file (the Unix *csh.1* manual entry)[40] transferred using the guidelines of the previous section (256 byte MTU or 216 byte MSS; 368 packets total). Compression

[39] The data volume from user side of a telnet is too small to benefit from data compression and can be adversely affected by the delay most compression algorithms (necessarily) add. The statistics and volume of the computer side of a telnet are similar to an (ASCII) FTP so these results should apply to either.

[40] The ten experiments described were each done on ten ASCII files (four long e-mail messages, three Unix

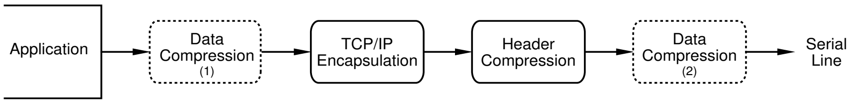

Figure 10: Data compression alternatives

ratios for the following ten tests are shown (reading left to right and top to bottom):

- data file (no compression or encapsulation)
- data $\rightarrow$ L–Z compressor
- data $\rightarrow$ TCP/IP encapsulation
- data $\rightarrow$ L–Z $\rightarrow$ TCP/IP
- data $\rightarrow$ TCP/IP $\rightarrow$ L–Z
- data $\rightarrow$ L–Z $\rightarrow$ TCP/IP $\rightarrow$ L–Z
- data $\rightarrow$ TCP/IP $\rightarrow$ Hdr. Compress.
- data $\rightarrow$ L–Z $\rightarrow$ TCP/IP $\rightarrow$ Hdr. Compress.
- data $\rightarrow$ TCP/IP $\rightarrow$ Hdr. Compress. $\rightarrow$ L–Z
- data $\rightarrow$ L–Z $\rightarrow$ TCP/IP $\rightarrow$ Hdr. Compress. $\rightarrow$ L–Z

	No data compress.	*L–Z on data*	*L–Z on wire*	*L–Z on both*
Raw Data	1.00	2.44	–	–
+ TCP Encap.	0.83	2.03	1.97	1.58
w/Hdr Comp.	0.98	2.39	2.26	1.66

Table 1: ASCII Text File Compression Ratios

The first column of table 1 says the data expands by 19% (‘compresses’ by .83) when encapsulated in TCP/IP and by 2% when encapsulated in header compressed TCP/IP.[41]

C source files and three Unix manual entries). The results were remarkably similar for different files and the general conclusions reached below apply to all ten files.

[41] This is what would be expected from the relative header sizes: 256/216 for TCP/IP and 219/216 for header compression.

The first row says L–Z compression is quite effective on this data, shrinking it to less than half its original size. Column four illustrates the well-known fact that it is a mistake to L–Z compress already compressed data. The interesting information is in rows two and three of columns two and three. These columns say that the benefit of data compression overwhelms the cost of encapsulation, even for straight TCP/IP. They also say that it is slightly better to compress the data before encapsulating it rather than compressing at the framing/modem level. The differences however are small — 3% and 6%, respectively, for the TCP/IP and header compressed encapsulations.[42]

Table 2 shows the same experiment for a 122,880 byte binary file (the Sun-3 *ps* executable). Although the raw data doesn't compress nearly as well, the results are qualitatively the same as for the ASCII data. The one significant change is in row two: It is about 3% better to compress the data in the modem rather than at the source if doing TCP/IP encapsulation (apparently, Sun binaries and TCP/IP headers have similar statistics). However, with header compression (row three) the results were similar to the ASCII data — it's about 3% worse to compress at the modem rather than the source.[43]

	No data compress.	*L-Z on data*	*L-Z on wire*	*L-Z on both*
Raw Data	1.00	1.72	–	–
+ TCP Encap.	0.83	1.43	1.48	1.21
w/Hdr Comp.	0.98	1.69	1.64	1.28

Table 2: Binary File Compression Ratios

[42] The differences are due to the wildly different byte patterns of TCP/IP datagrams and ASCII text. Any compression scheme with an underlying, Markov source model, such as Lempel-Ziv, will do worse when radically different sources are interleaved. If the relative proportions of the two sources are changed, i.e., the MTU is increased, the performance difference between the two compressor locations decreases. However, the rate of decrease is very slow — increasing the MTU by 400% (256 to 1024) only changed the difference between the data and modem L–Z choices from 2.5% to 1.3%.

[43] There are other good reasons to compress at the source: Far fewer packets have to be encapsulated and far fewer characters have to be sent to the modem. The author suspects that the 'compress data in the modem' alternative should be avoided except when faced with an intractable, vendor proprietary operating system.

Machine	*Average per-packet processing time (μsec.)*	
	Compress	*Decompress*
Sparcstation-1	24	18
Sun 4/260	46	20
Sun 3/60	90	90
Sun 3/50	130	150
HP9000/370	42	33
HP9000/360	68	70
DEC 3100	27	25
Vax 780	430	300
Vax 750	800	500
CCI Tahoe	110	140

Table 3: Compression code timings

6 Performance measurements

An implementation goal of compression code was to arrive at something simple enough to run at ISDN speeds (64Kbps) on a typical 1989 workstation. 64Kbps is a byte every $122\mu s$ so $120\mu s$ was (arbitrarily) picked as the target compression/decompression time.[44]

As part of the compression code development, a trace-driven exerciser was developed. This was initially used to compare different compression protocol choices then later to test the code on different computer architectures and do regression tests after performance 'improvements'. A small modification of this test program resulted in a useful measurement tool.[45] Table 3 shows the result of timing the compression code on all the machines available to the author (times were measured using a mixed telnet/ftp traffic trace). With the exception of the Vax architectures, which suffer from (a) having bytes in the wrong order and (b) a lousy compiler (Unix pcc), all machines essentially met the $120\mu s$ goal.

[44] The time choice wasn't completely arbitrary: Decompression is often done during the inter-frame 'flag' character time so, on systems where the decompression is done at the same priority level as the serial line input interrupt, times much longer than a character time would result in receiver overruns. And, with the current average of five byte frames (on the wire, including both compressed header and framing), a compression/decompression that takes one byte time can use at most 20% of the available time. This seems like a comfortable budget.

[45] Both the test program and timer program are included in the ftp-able package described in appendix A as files *tester.c* and *timer.c*.

7 Acknowlegements

The author is grateful to the members of the Internet Engineering Task Force, chaired by Phill Gross, who provided encouragement and thoughtful review of this work. Several patient beta-testers, particularly Sam Leffler and Craig Leres, tracked down and fixed problems in the initial implementation. Cynthia Livingston and Craig Partridge carefully read and greatly improved an unending sequence of partial drafts of this document. And last but not least, Telebit modem corporation, particularly Mike Ballard, encouraged this work from its inception and has been an ongoing champion of serial line and dial-up IP.

References

[1] BINGHAM, J. A. C. *Theory and Practice of Modem Design*. John Wiley & Sons, 1988.

[2] CAREY, M. B., CHAN, H.-T., DESCLOUX, A., INGLE, J. F., AND PARK, K. I. 1982/83 end office connection study: Analog voice and voiceband data transmission performance characterization of the public switched network. *Bell System Technical Journal 63*, 9 (Nov. 1984).

[3] CHIAPPA, N., 1988. Private communication.

[4] CLARK, D. D. The design philosophy of the DARPA Internet protocols. In *Proceedings of SIGCOMM '88* (Stanford, CA, Aug. 1988), ACM.

[5] FARBER, D. J., DELP, G. S., AND CONTE, T. M. *A Thinwire Protocol for connecting personal computers to the Internet*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Sept. 1984. RFC-914.

[6] KENT, C. A., AND MOGUL, J. Fragmentation considered harmful. In *Proceedings of SIGCOMM '87* (Aug. 1987), ACM.

[7] LEMPEL, A., AND ZIV, J. Compression of individual sequences via variable-rate encoding. *IEEE Transactions on Information Theory IT-24*, 5 (June 1978).

[8] NAGLE, J. *Congestion Control in IP/TCP Internetworks*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Jan. 1984. RFC-896.

[9] PERKINS, D. *Point-to-Point Protocol: A proposal for multi-protocol transmission of datagrams over point-to-point links*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Nov. 1989. RFC-1134.

[10] POSTEL, J., Ed. *Internet Protocol Specification*. SRI International, Menlo Park, CA, Sept. 1981. RFC-791.

[11] POSTEL, J., Ed. *Transmission Control Protocol Specification*. SRI International, Menlo Park, CA, Sept. 1981. RFC-793.

[12] ROMKEY, J. *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: Slip*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, June 1988. RFC-1055.

[13] SALTHOUSE, T. A. The skill of typing. *Scientific American 250*, 2 (Feb. 1984), 128–135.

[14] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984).

[15] SHNEIDERMAN, B. *Designing the User Interface*. Addison-Wesley, 1987.

A Sample Implementation

The following is a sample implementation of the protocol described in this document.

Since many people who might have the deal with this code are familiar with the Berkeley Unix kernel and its coding style (affectionately known as *kernel normal form*), this code was done in that style. It uses the Berkeley "subroutines" (actually, macros and/or inline assembler expansions) for converting to/from network byte order and copying/comparing strings of bytes. These routines are briefly described in sec. A.5 for anyone not familiar with them.

This code has been run on all the machines listed in the table on page 25. Thus, the author hopes there are no byte order or alignment problems (although there are embedded assumptions about alignment that are valid for Berkeley Unix but may not be true for other IP implementations — see the comments mentioning alignment in *sl_compress_tcp* and *sl_decompress_tcp*).

There was some attempt to make this code efficient. Unfortunately, that may have made portions of it incomprehensible. The author apologizes for any frustration this engenders. (In honesty, my C style is known to be obscure and claims of "efficiency" are simply a convenient excuse.)

This sample code and a complete Berkeley Unix implementation is available in machine readable form via anonymous ftp from Internet host ftp.ee.lbl.gov (128.3.254.68), file **cslip.tar.Z**. This is a compressed Unix tar file. It must be ftped in binary mode.

All of the code in this appendix is covered by the following copyright:

A.1 Definitions and State Data

```c
#define MAX_STATES 16  /* must be >2 and <255 */
#define MAX_HDR 128    /* max TCP + IP hdr length (from protocol def) */

/* packet types */
#define TYPE_IP 0x40
#define TYPE_UNCOMPRESSED_TCP 0x70
#define TYPE_COMPRESSED_TCP 0x80
#define TYPE_ERROR 0x00                  /* this is not a type that ever appears
                                          * on the wire.  The receive framer uses
                                          * it to tell the decompressor there was
                                          * a packet transmission error. */

/* Bits in first octet of compressed packet */
#define NEW_C   0x40     /* flag bits for what changed in a packet */
#define NEW_I   0x20
#define TCP_PUSH_BIT 0x10

#define NEW_S   0x08
#define NEW_A   0x04
#define NEW_W   0x02
#define NEW_U   0x01

/* reserved, special-case values of above */
#define SPECIAL_I (NEW_S|NEW_W|NEW_U)        /* echoed interactive traffic */
#define SPECIAL_D (NEW_S|NEW_A|NEW_W|NEW_U) /* unidirectional data */
#define SPECIALS_MASK (NEW_S|NEW_A|NEW_W|NEW_U)


/* "state" data for each active tcp conversation on the wire.  This
 * is basically a copy of the entire IP/TCP header from the last packet together
 * with a small identifier the transmit & receive ends of the line use to locate
 * saved header. */
struct cstate {
        struct cstate *cs_next;     /* next most recently used cstate (xmit only) */
        u_short cs_hlen;            /* size of hdr (receive only) */
        u_char cs_id;               /* connection # associated with this state */
        u_char cs_filler;
        union {
                char hdr[MAX_HDR];
                struct ip csu_ip;           /* ip/tcp hdr from most recent packet */
        } slcs_u;
};
#define cs_ip slcs_u.csu_ip
#define cs_hdr slcs_u.csu_hdr

/* all the state data for one serial line (we need one of these per line). */
struct slcompress {
        struct cstate *last_cs;                 /* most recently used tstate */
        u_char last_recv;                       /* last rcvd conn. id */
        u_char last_xmit;                       /* last sent conn. id */
```

```
        u_short flags;
        struct cstate tstate[MAX_STATES];       /* xmit connection states */
        struct cstate rstate[MAX_STATES];       /* receive connection states */
};
/* flag values */
#define SLF_TOSS 1              /* tossing rcvd frames because of input err */

/*
 * The following macros are used to encode and decode numbers.  They
 * all  assume that `cp' points to a buffer where the next byte
 * encoded (decoded) is to be stored (retrieved).  Since the decode
 * routines do arithmetic, they have to convert from and to network
 * byte order.
 */

/* ENCODE encodes a number that is known to be non-zero.  ENCODEZ
 * checks for zero (zero has to be encoded in the long, 3 byte form). */
#define ENCODE(n) { \
        if ((u_short)(n) >= 256) { \
                *cp++ = 0; \
                cp[1] = (n); \
                cp[0] = (n) >> 8; \
                cp += 2; \
        } else { \
                *cp++ = (n); \
        } \
}
#define ENCODEZ(n) { \
        if ((u_short)(n) >= 256 || (u_short)(n) == 0) { \
                *cp++ = 0; \
                cp[1] = (n); \
                cp[0] = (n) >> 8; \
                cp += 2; \
        } else { \
                *cp++ = (n); \
        } \
}

/* DECODEL takes the (compressed) change at byte cp and adds it to
 * the current value of packet field 'f' (which must be a 4-byte
 * (long) integer in network byte order).  DECODES does the same
 * for a 2-byte (short) field.  DECODEU takes the change at cp and stuffs
 * it into the (short) field f.  'cp' is updated to point to the
 * next field in the compressed header. */
#define DECODEL(f) { \
        if (*cp == 0) {\
                (f) = htonl(ntohl(f) + ((cp[1] << 8) | cp[2])); \
                cp += 3; \
        } else { \
                (f) = htonl(ntohl(f) + (u_long)*cp++); \
        } \
}
```

```
#define DECODES(f) { \
        if (*cp == 0) {\
                (f) = htons(ntohs(f) + ((cp[1] << 8) | cp[2])); \
                cp += 3; \
        } else { \
                (f) = htons(ntohs(f) + (u_long)*cp++); \
        } \
}
#define DECODEU(f) { \
        if (*cp == 0) {\
                (f) = htons((cp[1] << 8) | cp[2]); \
                cp += 3; \
        } else { \
                (f) = htons((u_long)*cp++); \
        } \
}
```

A.2 Compression

This routine looks daunting but isn't really. The code splits into four approximately equal sized sections: The first quarter manages a circularly linked, least-recently-used list of "active" TCP connections.[46] The second figures out the sequence/ack/window/urg changes and builds the bulk of the compressed packet. The third handles the special-case encodings. The last quarter does packet ID and connection ID encoding and replaces the original packet header with the compressed header.

The arguments to this routine are a pointer to a packet to be compressed, a pointer to the compression state data for the serial line, and a flag which enables or disables connection id (C bit) compression.

Compression is done "in-place" so, if a compressed packet is created, both the start address and length of the incoming packet (the *off* and *len* fields of *m*) will be updated to reflect the removal of the original header and its replacement by the compressed header. If either a compressed or uncompressed packet is created, the compression state is updated. This routines returns the packet type for the transmit framer (TYPE_IP, TYPE_UNCOMPRESSED_TCP or TYPE_COMPRESSED_TCP).

Because 16 and 32 bit arithmetic is done on various header fields, the incoming IP packet must be aligned appropriately (e.g., on a SPARC, the IP header is aligned on a 32-bit boundary). Substantial changes would have to be made to the code below if this were not true (and it would probably be cheaper to byte copy the incoming header to somewhere correctly aligned than to make those changes).

Note that the outgoing packet will be aligned arbitrarily (e.g., it could easily start on an odd-byte boundary).

[46] The two most common operations on the connection list are a 'find' that terminates at the first entry (a new packet for the most recently used connection) and moving the last entry on the list to the head of the list (the first packet from a new connection). A circular list efficiently handles these two operations.

```
u_char
sl_compress_tcp(m, comp, compress_cid)
        struct mbuf *m;
        struct slcompress *comp;
        int compress_cid;
{
        register struct cstate *cs = comp->last_cs->cs_next;
        register struct ip *ip = mtod(m, struct ip *);
        register u_int hlen = ip->ip_hl;
        register struct tcphdr *oth;        /* last TCP header */
        register struct tcphdr *th;         /* current TCP header */
        register u_int deltaS, deltaA;      /* general purpose temporaries */
        register u_int changes = 0;         /* change mask */
        u_char new_seq[16];                 /* changes from last to current */
        register u_char *cp = new_seq;

        /* Bail if this is an IP fragment or if the TCP packet isn't
         * `compressible' (i.e., ACK isn't set or some other control bit is
         * set).  (We assume that the caller has already made sure the
         * packet is IP proto TCP).  */
        if ((ip->ip_off & htons(0x3fff)) || m->m_len < 40)
                return (TYPE_IP);

        th = (struct tcphdr *)&((int *)ip)[hlen];
        if ((th->th_flags & (TH_SYN|TH_FIN|TH_RST|TH_ACK)) != TH_ACK)
                return (TYPE_IP);

        /* Packet is compressible -- we're going to send either a
         * COMPRESSED_TCP or UNCOMPRESSED_TCP packet.  Either way we need
         * to locate (or create) the connection state.  Special case the
         * most recently used connection since it's most likely to be used
         * again & we don't have to do any reordering if it's used.  */
        if (ip->ip_src.s_addr != cs->cs_ip.ip_src.s_addr ||
            ip->ip_dst.s_addr != cs->cs_ip.ip_dst.s_addr ||
            *(int *)th != ((int *)&cs->cs_ip)[cs->cs_ip.ip_hl]) {

                /* Wasn't the first -- search for it.
                 *
                 * States are kept in a circularly linked list with
                 * last_cs pointing to the end of the list.  The
                 * list is kept in lru order by moving a state to the
                 * head of the list whenever it is referenced.  Since
                 * the list is short and, empirically, the connection
                 * we want is almost always near the front, we locate
                 * states via linear search.  If we don't find a state
                 * for the datagram, the oldest state is (re-)used. */
                register struct cstate *lcs;
                register struct cstate *lastcs = comp->last_cs;

                do {
                        lcs = cs; cs = cs->cs_next;
                        if (ip->ip_src.s_addr == cs->cs_ip.ip_src.s_addr
```

```
                        && ip->ip_dst.s_addr == cs->cs_ip.ip_dst.s_addr
                        && *(int *)th == ((int *)&cs->cs_ip)[cs->cs_ip.ip_hl])
                            goto found;
        } while (cs != lastcs);

        /* Didn't find it -- re-use oldest cstate.  Send an
         * uncompressed packet that tells the other side what
         * connection number we're using for this conversation.
         * Note that since the state list is circular, the oldest
         * state points to the newest and we only need to set
         * last_cs to update the lru linkage. */
        comp->last_cs = lcs;
        hlen += th->th_off;
        hlen <<= 2;
        goto uncompressed;

found:
        /* Found it -- move to the front on the connection list. */
        if (lastcs == cs)
                comp->last_cs = lcs;
        else {
                lcs->cs_next = cs->cs_next;
                cs->cs_next = lastcs->cs_next;
                lastcs->cs_next = cs;
        }
}

/* Make sure that only what we expect to change changed. The first
 * line of the 'if' checks the IP protocol version, header length &
 * type of service.  The 2nd line checks the "Don't fragment" bit.
 * The 3rd line checks the time-to-live and protocol (the protocol
 * check is unnecessary but costless).  The 4th line checks the TCP
 * header length.  The 5th line checks IP options, if any.  The 6th
 * line checks TCP options, if any.  If any of these things are
 * different between the previous & current datagram, we send the
 * current datagram 'uncompressed'.  */
oth = (struct tcphdr *)&((int *)&cs->cs_ip)[hlen];
deltaS = hlen;
hlen += th->th_off;
hlen <<= 2;

if (((u_short *)ip)[0] != ((u_short *)&cs->cs_ip)[0] ||
    ((u_short *)ip)[3] != ((u_short *)&cs->cs_ip)[3] ||
    ((u_short *)ip)[4] != ((u_short *)&cs->cs_ip)[4] ||
    th->th_off != oth->th_off ||
    (deltaS > 5 && BCMP(ip + 1, &cs->cs_ip + 1, (deltaS - 5) << 2)) ||
    (th->th_off > 5 && BCMP(th + 1, oth + 1, (th->th_off - 5) << 2)))
        goto uncompressed;

/* Figure out which of the changing fields changed.  The receiver
 * expects changes in the order: urgent, window, ack, seq. */
if (th->th_flags & TH_URG) {
```

```
        deltaS = ntohs(th->th_urp);
        ENCODEZ(deltaS);
        changes |= NEW_U;
} else if (th->th_urp != oth->th_urp)
        /* argh! URG not set but urp changed -- a sensible
         * implementation should never do this but RFC793 doesn't
         * prohibit the change so we have to deal with it. */
         goto uncompressed;

if (deltaS = (u_short)(ntohs(th->th_win) - ntohs(oth->th_win))) {
        ENCODE(deltaS);
        changes |= NEW_W;
}
if (deltaA = ntohl(th->th_ack) - ntohl(oth->th_ack)) {
        if (deltaA > 0xffff)
                goto uncompressed;
        ENCODE(deltaA);
        changes |= NEW_A;
}
if (deltaS = ntohl(th->th_seq) - ntohl(oth->th_seq)) {
        if (deltaS > 0xffff)
                goto uncompressed;
        ENCODE(deltaS);
        changes |= NEW_S;
}

/* Look for the special-case encodings. */
switch(changes) {

case 0:
        /* Nothing changed. If this packet contains data and the
         * last one didn't, this is probably a data packet following
         * an ack (normal on an interactive connection) and we send
         * it compressed.  Otherwise it's probably a retransmit,
         * retransmitted ack or window probe.  Send it uncompressed
         * in case the other side missed the compressed version.  */
        if (ip->ip_len != cs->cs_ip.ip_len && ntohs(cs->cs_ip.ip_len) == hlen)
                break;

        /* (fall through) */

case SPECIAL_I:
case SPECIAL_D:
        /* actual changes match one of our special case encodings --
         * send packet uncompressed.  */
        goto uncompressed;

case NEW_S|NEW_A:
        if (deltaS == deltaA && deltaS == ntohs(cs->cs_ip.ip_len) - hlen) {
                /* special case for echoed terminal traffic */
                changes = SPECIAL_I;
                cp = new_seq;
```

```
		}
		break;

	case NEW_S:
		if (deltaS == ntohs(cs->cs_ip.ip_len) - hlen) {
			/* special case for data xfer */
			changes = SPECIAL_D;
			cp = new_seq;
		}
		break;
	}
	deltaS = ntohs(ip->ip_id) - ntohs(cs->cs_ip.ip_id);
	if (deltaS != 1) {
		ENCODEZ(deltaS);
		changes |= NEW_I;
	}
	if (th->th_flags & TH_PUSH)
		changes |= TCP_PUSH_BIT;
	/* Grab the cksum before we overwrite it below.  Then update our
	 * state with this packet's header. */
	deltaA = ntohs(th->th_sum);
	BCOPY(ip, &cs->cs_ip, hlen);

	/* We want to use the original packet as our compressed packet.
	 * (cp - new_seq) is the number of bytes we need for compressed
	 * sequence numbers.  In addition we need one byte for the change
	 * mask, one for the connection id and two for the tcp checksum.
	 * So, (cp - new_seq) + 4 bytes of header are needed.  hlen is how
	 * many bytes of the original packet to toss so subtract the two to
	 * get the new packet size.  */
	deltaS = cp - new_seq;
	cp = (u_char *)ip;
	if (compress_cid == 0 || comp->last_xmit != cs->cs_id) {
		comp->last_xmit = cs->cs_id;
		hlen -= deltaS + 4;
		cp += hlen;
		*cp++ = changes | NEW_C;
		*cp++ = cs->cs_id;
	} else {
		hlen -= deltaS + 3;
		cp += hlen;
		*cp++ = changes;
	}
	m->m_len -= hlen;
	m->m_off += hlen;
	*cp++ = deltaA >> 8;
	*cp++ = deltaA;
	BCOPY(new_seq, cp, deltaS);
	return (TYPE_COMPRESSED_TCP);

	/* Update connection state cs & send uncompressed packet ('uncompressed'
	 * means a regular ip/tcp packet but with the 'conversation id' we hope
```

```
         * to use on future compressed packets in the protocol field). */
uncompressed:
        BCOPY(ip, &cs->cs_ip, hlen);
        ip->ip_p = cs->cs_id;
        comp->last_xmit = cs->cs_id;
        return (TYPE_UNCOMPRESSED_TCP);
}
```

A.3 Decompression

This routine decompresses a received packet. It is called with a pointer to the packet, the packet length and type, and a pointer to the compression state structure for the incoming serial line. It returns a pointer to the resulting packet or zero if there were errors in the incoming packet. If the packet is COMPRESSED_TCP or UNCOMPRESSED_TCP, the compression state will be updated.

The new packet will be constructed in-place. That means that there must be 128 bytes of free space in front of bufp to allow room for the reconstructed IP and TCP headers. The reconstructed packet will be aligned on a 32-bit boundary.

```
u_char *
sl_uncompress_tcp(bufp, len, type, comp)
        u_char *bufp;
        int len;
        u_int type;
        struct slcompress *comp;
{
        register u_char *cp;
        register u_int hlen, changes;
        register struct tcphdr *th;
        register struct cstate *cs;
        register struct ip *ip;

        switch (type) {

        case TYPE_ERROR:
        default:
                goto bad;

        case TYPE_IP:
                return (bufp);

        case TYPE_UNCOMPRESSED_TCP:
                /* Locate the saved state for this connection.  If the
                 * state index is legal, clear the 'discard' flag. */
                ip = (struct ip *) bufp;
                if (ip->ip_p >= MAX_STATES)
                        goto bad;

                cs = &comp->rstate[comp->last_recv = ip->ip_p];
```

```
                comp->flags &= ~SLF_TOSS;
                /* Restore the IP protocol field then save a copy of this packet
                 * header.  (The checksum is zeroed in the copy so we don't
                 * have to zero it each time we process a compressed packet. */
                ip->ip_p = IPPROTO_TCP;
                hlen = ip->ip_hl;
                hlen += ((struct tcphdr *)&((int *)ip)[hlen])->th_off;
                hlen <<= 2;
                BCOPY(ip, &cs->cs_ip, hlen);
                cs->cs_ip.ip_sum = 0;
                cs->cs_hlen = hlen;
                return (bufp);

        case TYPE_COMPRESSED_TCP:
                break;
        }
        /* We've got a compressed packet. */
        cp = bufp;
        changes = *cp++;
        if (changes & NEW_C) {
                /* Make sure the state index is in range, then grab the state.
                 * If we have a good state index, clear the 'discard' flag. */
                if (*cp >= MAX_STATES)
                        goto bad;

                comp->flags &= ~SLF_TOSS;
                comp->last_recv = *cp++;
        } else {
                /* This packet has an implicit state index.  If we've
                 * had a line error since the last time we got an
                 * explicit state index, we have to toss the packet. */
                if (comp->flags & SLF_TOSS)
                        return ((u_char *)0);
        }
        /* Find the state then fill in the TCP checksum and PUSH bit. */
        cs = &comp->rstate[comp->last_recv];
        hlen = cs->cs_ip.ip_hl << 2;
        th = (struct tcphdr *)&((u_char *)&cs->cs_ip)[hlen];
        th->th_sum = htons((*cp << 8) | cp[1]);
        cp += 2;
        if (changes & TCP_PUSH_BIT)
                th->th_flags |= TH_PUSH;
        else
                th->th_flags &= ~TH_PUSH;

        /* Fix up the state's ack, seq, urg and win fields based on the changemask. */
        switch (changes & SPECIALS_MASK) {
        case SPECIAL_I:
                {
                register u_int i = ntohs(cs->cs_ip.ip_len) - cs->cs_hlen;
                th->th_ack = htonl(ntohl(th->th_ack) + i);
                th->th_seq = htonl(ntohl(th->th_seq) + i);
```

```
                }
                break;

        case SPECIAL_D:
                th->th_seq = htonl(ntohl(th->th_seq) + ntohs(cs->cs_ip.ip_len) - cs->cs_hlen);
                break;

        default:
                if (changes & NEW_U) {
                        th->th_flags |= TH_URG;
                        DECODEU(th->th_urp)
                } else
                        th->th_flags &=~ TH_URG;
                if (changes & NEW_W)
                        DECODES(th->th_win)
                if (changes & NEW_A)
                        DECODEL(th->th_ack)
                if (changes & NEW_S)
                        DECODEL(th->th_seq)
                break;
        }
        /* Update the IP ID */
        if (changes & NEW_I)
                DECODES(cs->cs_ip.ip_id)
        else
                cs->cs_ip.ip_id = htons(ntohs(cs->cs_ip.ip_id) + 1);

        /* At this point, cp points to the first byte of data in the
         * packet.  If we're not aligned on a 4-byte boundary, copy the
         * data down so the IP & TCP headers will be aligned.  Then back up
         * cp by the TCP/IP header length to make room for the reconstructed
         * header (we assume the packet we were handed has enough space to
         * prepend 128 bytes of header).  Adjust the lenth to account for
         * the new header & fill in the IP total length.
         */
        len -= (cp - bufp);
        if (len < 0)
                /* we must have dropped some characters (crc should detect
                 * this but the old slip framing won't) */
                goto bad;

        if ((int)cp & 3) {
                if (len > 0)
                        OVBCOPY(cp, (int)cp &~ 3, len);
                cp = (u_char *)((int)cp &~ 3);
        }
        cp -= cs->cs_hlen;
        len += cs->cs_hlen;
        cs->cs_ip.ip_len = htons(len);
        BCOPY(&cs->cs_ip, cp, cs->cs_hlen);

        /* recompute the ip header checksum */
```

```
	{
		register u_short *bp = (u_short *)cp;
		for (changes = 0; hlen > 0; hlen -= 2)
			changes += *bp++;
		changes = (changes & 0xffff) + (changes >> 16);
		changes = (changes & 0xffff) + (changes >> 16);
		((struct ip *)cp)->ip_sum = ~ changes;
	}
	return (cp);

bad:
	comp->flags |= SLF_TOSS;
	return ((u_char *)0);
}
```

A.4 Initialization

This routine initializes the state structure for both the transmit and receive halves of some serial line. It must be called each time the line is brought up.

```
void
sl_compress_init(comp)
	struct slcompress *comp;
{
	register u_int i;
	register struct cstate *tstate = comp->tstate;

	bzero((char *)comp, sizeof(*comp));
	for (i = MAX_STATES - 1; i > 0; --i) {
		tstate[i].cs_id = i;
		tstate[i].cs_next = &tstate[i - 1];
	}
	tstate[0].cs_next = &tstate[MAX_STATES - 1];
	tstate[0].cs_id = 0;
	comp->last_cs = &tstate[0];
	comp->last_recv = 255;
	comp->last_xmit = 255;
}
```

A.5 Berkeley Unix dependencies

Note: The following is of interest only if you are trying to bring the sample code up on a system that is not derived from 4BSD (Berkeley Unix).

The code uses the normal Berkeley Unix header files (from /usr/include/netinet) for definitions of the structure of IP and TCP headers. The structure tags tend to follow the protocol RFCs closely and should be obvious even if you do not have access to a 4BSD system.[47]

The macro *BCOPY(src, dst, amt)* is invoked to copy *amt* bytes from *src* to *dst*. In BSD, it translates into a call to *bcopy*. If you have the misfortune to be running System-V Unix, it can be translated into a call to *memcpy*. The macro *OVBCOPY(src, dst, amt)* is used to copy when *src* and *dst* overlap (i.e., when doing the 4-byte alignment copy). In the BSD kernel, it translates into a call to *ovbcopy*. Since AT&T botched the definition of *memcpy*, this should probably translate into a copy loop under System-V.

The macro *BCMP(src, dst, amt)* is invoked to compare *amt* bytes of *src* and *dst* for equality. In BSD, it translates into a call to *bcmp*. In System-V, it can be translated into a call to *memcmp* or you can write a routine to do the compare. The routine should return zero if all bytes of *src* and *dst* are equal and non-zero otherwise.

The routine *ntohl(dat)* converts (4 byte) long *dat* from network byte order to host byte order. On a reasonable cpu this can be the no-op macro:

```
#define ntohl(dat) (dat)
```

On a Vax or IBM PC (or anything with Intel byte order), you will have to define a macro or routine to rearrange bytes.

The routine *ntohs(dat)* is like *ntohl* but converts (2 byte) shorts instead of longs. The routines *htonl(dat)* and *htons(dat)* do the inverse transform (host to network byte order) for longs and shorts.

A *struct mbuf* is used in the call to *sl_compress_tcp* because that routine needs to modify both the start address and length if the incoming packet is compressed. In BSD, an mbuf is the kernel's buffer management structure. If other systems, the following definition should be sufficient:

```
struct mbuf {
        u_char  *m_off;  /* pointer to start of data */
        int     m_len;   /* length of data */
};

#define mtod(m, t) ((t)(m->m_off))
```

[47] In the event they are not obvious, the header files (and all the Berkeley networking code) can be anonymous ftp'd from host ucbarpa.berkeley.edu, files pub/4.3/tcp.tar and pub/4.3/inet.tar.

B Compatibility with past mistakes

When combined with the modern PPP serial line protocol[9], the use of header compression is automatic and invisible to the user. Unfortunately, many sites have existing users of the SLIP described in [12] which doesn't allow for different protocol types to distinguish header compressed packets from IP packets or for version numbers or an option exchange that could be used to automatically negotiate header compression.

The author has used the following tricks to allow header compressed SLIP to interoperate with the existing servers and clients. Note that these are *hacks* for compatibility with past mistakes and should be offensive to any right thinking person. They are offered solely to ease the pain of running SLIP while users wait patiently for vendors to release PPP.

B.1 Living without a framing 'type' byte

The bizarre packet type numbers in sec. A.1 were chosen to allow a 'packet type' to be sent on lines where it is undesirable or impossible to add an explicit type byte. Note that the first byte of an IP packet always contains "4" (the IP protocol version) in the top four bits. And that the most significant bit of the first byte of the compressed header is ignored. Using the packet types in sec. A.1, the type can be encoded in the most significant bits of the outgoing packet using the code

```
p->dat[0] |= sl_compress_tcp(p, comp);
```

and decoded on the receive side by

```
if (p->dat[0] & 0x80)
        type = TYPE_COMPRESSED_TCP;
else if (p->dat[0] >= 0x70) {
        type = TYPE_UNCOMPRESSED_TCP;
        p->dat[0] &=~ 0x30;
} else
        type = TYPE_IP;
status = sl_uncompress_tcp(p, type, comp);
```

B.2 Backwards compatible SLIP servers

The SLIP described in [12] doesn't include any mechanism that could be used to automatically negotiate header compression. It would be nice to allow users of this SLIP to use header compression but, when users of the two SLIP varients share a common server, it would be annoying and difficult to manually configure both ends of each connection to enable compression. The following procedure can be used to avoid manual configuration.

Since there are two types of dial-in clients (those that implement compression and those that don't) but one server for both types, it's clear that the server will be reconfiguring for each new client session but clients change configuration seldom if ever. If manual configuration has to be done, it should be done on the side that changes infrequently — the client. This suggests that the server should somehow learn from the client whether to use header compression. Assuming symmetry (i.e., if compression is used at all it should be used both directions) the server can use the receipt of a compressed packet from some client to indicate that it can send compressed packets to that client. This leads to the following algorithm:

There are two bits per line to control header compression: *allowed* and *on*. If *on* is set, compressed packets are sent, otherwise not. If *allowed* is set, compressed packets can be received and, if an UNCOMPRESSED_TCP packet arrives when *on* is clear, *on* will be set.[48] If a compressed packet arrives when *allowed* is clear, it will be ignored.

Clients are configured with both bits set (*allowed* is always set if *on* is set) and the server starts each session with *allowed* set and *on* clear. The first compressed packet from the client (which must be a UNCOMPRESSED_TCP packet) turns on compression for the server.

C More aggressive compression

As noted in sec. 3.2.2, easily detected patterns exist in the stream of compressed headers, indicating that more compression could be done. Would this be worthwhile?

The average compressed datagram has only seven bits of header.[49] The framing must be at least one bit (to encode the 'type') and will probably be more like two to three bytes. In most interesting cases there will be at least one byte of data. Finally, the end-to-end check—the TCP checksum—must be passed through unmodified.[50]

The framing, data and checksum will remain even if the header is completely compressed out so the change in average packet size is, at best, four bytes down to three bytes and one bit — roughly a 25% improvement in delay.[51] While this may seem significant, on a 2400 bps line it means that typing echo response takes 25 rather than 29 ms. At the present stage of human evolution, this difference is not detectable.

[48] Since [12] framing doesn't include error detection, one should be careful not to 'false trigger' compression on the server. The UNCOMPRESSED_TCP packet should checked for consistency (e.g., IP checksum correctness) before compression is enabled. Arrival of COMPRESSED_TCP packets should not be used to enable compression.

[49] Tests run with several million packets from a mixed traffic load (i.e., statistics kept on a year's traffic from my home to work) show that 80% of packets use one of the two special encodings and, thus, the only header is the change mask.

[50] If someone tries to sell you a scheme that compresses the TCP checksum "Just say no". Some poor fool has yet to have the sad experience that reveals the *end-to-end argument* is gospel truth. Worse, since the fool is subverting *your* end-to-end error check, *you* may pay the price for this education and they will be none the wiser. What does it profit a man to gain two byte times of delay and lose peace of mind?

[51] Note again that we must be concerned about interactive delay to be making this argument: Bulk data transfer performance will be dominated by the time to send the data and the difference between three and four byte headers on a datagram containing tens or hundreds of data bytes is, practically, no difference.

However, the author sheepishly admits to perverting this compression scheme for a very special case data-acquisition problem: We had an instrument and control package floating at 200KV, communicating with ground level via a telemetry system. For many reasons (multiplexed communication, pipelining, error recovery, availability of well tested implementations, etc.), it was convenient to talk to the package using TCP/IP. However, since the primary use of the telemetry link was data acquisition, it was designed with an uplink channel capacity $<0.5\%$ the downlink's. To meet application delay budgets, data packets were 100 bytes and, since TCP acks every other packet, the relative uplink bandwidth for acks is $a/200$ where a is the total size of ack packets. Using the scheme in this paper, the smallest ack is four bytes which would imply an uplink bandwidth 2% of the downlink. This wasn't possible so we used the scheme described in footnote 14: If the first bit of the frame was one, it meant "same compressed header as last time". Otherwise the next two bits gave one of the types described in sec. 3.2. Since the link had excellent forward error correction and traffic made only a single hop, the TCP checksum was compressed out (blush!) of the "same header" packet types[52] so the total header size for these packets was one bit. Over several months of operation, more than 99% of the 40 byte TCP/IP headers were compressed down to one bit.[53]

D Security Considerations

Security considerations are not addressed in this memo.

E Author's address

Address: Van Jacobson
Real Time Systems Group
Mail Stop 46A
Lawrence Berkeley Laboratory
Berkeley, CA 94720

Phone: Use email (author ignores his phone)

EMail: van@helios.ee.lbl.gov

[52] The checksum was re-generated in the decompressor and, of course, the "toss" logic was made considerably more aggressive to prevent error propagation.

[53] We have heard the suggestion that "real-time" needs require abandoning TCP/IP in favor of a "light-weight" protocol with smaller headers. It is difficult to envision a protocol that averages less than one header bit per packet.