A Unified Header Compression Framework for Low-Bandwidth Links*

Jeremy Lilley, Jason Yang, Hari Balakrishnan, Srinivasan Seshan†
MIT Laboratory for Computer Science
Cambridge, MA 02139
{jjlilley, jcyang, hari}@lcs.mit.edu, sseshan@us.ibm.com

Abstract

Compressing protocol headers has traditionally been an attractive way of conserving bandwidth over low-speed links, including those in wireless systems. However, despite the growth in recent years in the number of end-to-end protocols beyond TCP/IP, header compression deployment for these protocols has not kept pace. This is in large part due to complexities in implementation, which often requires a detailed knowledge of kernel internals, and a lack of a common way of pursuing the general problem across a variety of end-to-end protocols. To address this, rather than defining several new protocol-specific standards, we present a unified framework for header compression. This framework includes a simple, platform-independent header description language that protocol implementors can use to describe high-level header properties, and a platform-specific code generation tool that produces kernel source code automatically from this header specification. Together, the high-level description language and code generator free protocol designers from having to understand any details of the target platform, enabling them to implement header compression with relatively little effort. We analyze the performance of compression produced using this framework for TCP/IP in the Linux 2.0 kernel and demonstrate that unified, automatically-generated header compression without significant performance penalty is viable.

1 Introduction

Limited bandwidth is one of the fundamental challenges in mobile and wireless networking systems. Even as the current explosive growth in mobile systems encourages the deployment of more wireless infrastructure and channels, bandwidth remains scarce, particularly in comparison with processor or storage technologies. In addition, many mobile links have considerations such as power consumption or transceiver circuit complexity reduce their possible bandwidth beyond the raw limitations of the physical medium or appropriate radio regulatory restrictions.

But, at a time where wireless applications are becoming increasingly important, network protocols and particularly their headers are becoming more varied and more complex. Specialized and application-specific protocols (e.g., RTP/RTCP [29], RTSP [30], HTTP [9], NFS [25], AFS [14], SDP [12], RDP [26], RPC [34], Java RMI [35]), various tunneling protocols, and those of popular media players [28, 3], are proliferating as the networked world moves beyond its previous staples of bulk FTP transfers and telnet. Yet these headers can be bandwidth intensive; for example, the header bandwidth of an RTP streaming media transmission at 20ms intervals amounts to 16kbps, which can be significant for a low-bandwidth wide-area wireless link. Furthermore, relatively little consideration is often given in design to protocols' behavior over low-speed links because application designers are rightfully concerned about correctness and performance over conventional Internet paths.

In addition, established protocols are becoming more complex with the addition of IPv6 [4] and Mobile IP support [18]. The TCP/IP protocol header, which is 40 bytes, can add significant delays to interactive applications that use small packets. IPv6 increases the size of this header to 60 bytes, and with Mobile IPv6 encapsulation, the header grows to 100 bytes. Additional TCP options such as the timestamp option [16] contribute as well. Reducing this header size, which can be a significant per-packet penalty for mobile systems, goes a long way toward making low-bandwidth wireless applications more responsive and efficient. Header compression is a method to do this, taking advantage of the commonality of header fields across packets on any flow.

Thus, many end-to-end protocols will benefit from header compression over low-bandwidth links. However, designing and implementing this, particularly in the core of a networking stack, can be a daunting task. Making the changes usually requires making detailed modifications to the operating system kernel and adding various drivers. For example, adding a simple header compression algorithm to the Linux PPP driver required modifying well over 400 lines in various operating system files and daemon source code *before* a line of compression code could be written. This problem becomes even worse when there are many platforms to support. In addition, validating and re-checking assumptions is tedious, and the appropriate debugging tools are scarce. As a result, header com-

*This research was supported in part by DARPA (Grant No. MDA972-99-1-0014), NTT Corporation, and IBM.

†IBM TJ Watson Research Center, Hawthorne, NY 10532.

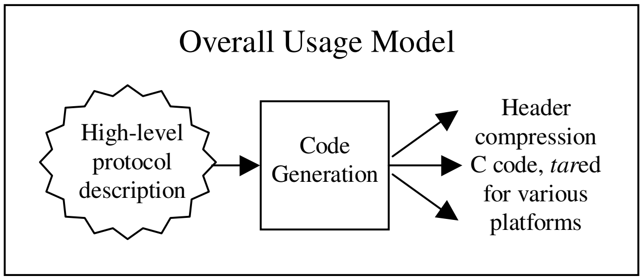

Figure 1: The general usage model in this framework: A high-level protocol description is given to the code generator tool, which produces all the applicable driver code changes.

pression schemes end up not being deployed in many places even when they may be beneficial.

Losses further complicate the picture. Any header compression scheme requires some state to be maintained at each end of the constrained link. Unfortunately, a simple compression scheme with little loss resilience may get its connection state corrupted from a lossy link and end up worse off than without the compression. In addition, if the link is fast enough, the possible savings may not be worth the extra error probability that any compression scheme could introduce. Header compression should only be performed when it is truly beneficial.

To address these problems, we have developed a unified approach to the header compression problem by providing a generic compression format and mechanism that may be tailored to various end-to-end protocols. Furthermore, we provide a set of tools that allows header compression code to be automatically generated and deployed from this high-level description of a protocol. This includes a simple protocol description language to characterize the predictability of the header fields, which is easily extensible to support any number of user-defined protocols. From the protocol descriptions, our tools automatically generate all the necessary kernel and driver source code for the target platform. Using standard header compression error-recovery algorithms, the code also provides a measure of protection against link losses. Finally, our results indicate that automatically generated header compression provides similar performance to hand-optimized solutions.

We demonstrate our system in a prototype implementation in the Point to Point Protocol (PPP) [31, 13] stack (version 2.3.5) of the Linux 2.0.36 operating system using the Redhat 5.2 distribution. The tools can be ported to accommodate other implementation platforms from the same protocol description. Even when a platform has not been targeted, the extended and unextended versions of the drivers are all interoperable due to the compression negotiation features in the link layer.

Given this, we envision that an organization wanting to improve performance over various low-bandwidth links with header compression would be able to quickly write or modify appropriate protocol descriptions for even obscure and specialized protocols it uses. End-to-end protocol designers only need to produce protocol-specific details of their header fields, specifying the properties of these fields in a readable language. As illustrated in Figure 1, they would then be able to automatically generate the appropriate driver files for each necessary platform by running the code generation scripts. This model produces a rapidly deployable header compression solution for a designed protocol, optimized for low-speed links, without requiring any platform-specific coding or extensive effort.

1.1 Goals

The major goal of our work is to provide and evaluate a toolkit for compressing protocol headers based on a high-level specification. This goals of this toolkit can be broken down into:

- Producing a generic, unified compression format that may be customized for different protocols.
- Providing a high-level language for characterizing important features necessary for a protocol-specific header compression algorithm.
- Creating a platform for freeing the protocol designer from having to deal with low-level OS kernel details.
- Ensuring that errors can be tolerated by the resulting protocol.
- Evaluating these generated protocols' network performance, as well as their use of the host machine both in terms of space and CPU efficiency.

The rest of the paper follows the outline of these goals. Section 2 discusses related work, Section 3 provides an overview of the compression scheme, and Section 4 discusses the header description language. Section 5 highlights error handling, while Sections 6, 7, 8, and 9 convey implementation, results, discussion, and our conclusion.

2 Related work

One of the earliest examples of header compression was the Thinwire protocol [8], proposed in 1984, which uses a simple 20-bit field to specify which of the packet header's first 20 bytes have changed. Thinwire was relatively protocol-neutral, but not optimized for common traffic.

In early 1990, Van Jacobson proposed a TCP/IP-specific compression algorithm with many optimizations taking advantage of this protocol's intricacies [17]. Sending between 3-5 bytes of the 40-byte header in the common case, VJ TCP header compression is efficient, and the most widely deployed header compression protocol. However, VJ compression only works with TCP/IP packets, and derives many of its strengths from its rigid specification.

Since then, specifications for the compression of a number of other protocols have been written. Degermark proposed additional compression algorithms for UDP/IP and TCP/IPv6 [5]. Detailed specifications for compressing these protocols, as well as others such as RTP [29], were described in a number of subsequent RFCs including 2507 [6], 2508 [1], and 2509 [7]. Each of these descriptions specify a solution for a given protocol. While any protocol can be characterized, building a new header compressor for that characterization must proceed from scratch every time. This impacts driver availability, and, indeed, no header compression algorithms have been very widely deployed since VJ TCP compression. We have leveraged many ideas from this previous research to build a generalized header compression tool.

3 Compression overview

Ideally, any header compression scheme will avoid sending redundant data, while protecting the protocol from extra errors caused by maintaining state over a potentially loss-prone link. In doing this, the predictable properties of the underlying protocol need to be identified and exploited. For example, VJ TCP header compression identifies enough redundancy in TCP/IP to compress 40-byte headers to as few as three bytes, including a two-byte checksum. As far as any generalized compression scheme is concerned, the efficiency will vary depending on the predictability of the protocol header and how well this predictability can be captured in a specification.

Since many useful categorizations for fields have been identified from previous work in header compression, we have chosen to use similar terminology and style. The basic idea is to identify each field in the protocol, which may be located at any bit offset and length in the frame, and then to associate various attributes with the field. While different schemes have been introduced for classifying fields, we found it convenient to use a derivative of some of the fields used by Degermark [5]. In particular, the four main attributes for fields in our scheme are *constant*, *delta*, *inferable*, and *random*:

- *Constant* fields do not change among the headers in a given protocol session, connection, or "*context*." These typically include the source and destination IP addresses.
- *Delta* elements represent fields that change by small quantities and may not change every time. An example of this field is the TCP sequence numbers, where the compressor could gain by sending the difference of the values between consecutive packets. Since deltas may not change every packet, a bit field can be used to send only non-zero changes.
- *Inferred* fields can be determined based other known information. This could include the packet length field, which can be derived from the underlying framing layer.
- *Random* fields, such as checksums, are elements with little regular pattern that are best sent unchanged.

After classifying the header fields, the main idea behind compression is to send only the information that is needed to reconstruct the header. In particular, only the delta and random fields need to be sent per packet in the common case. Inferable fields need a formula to derive the proper field value, but this does not require extra data to be sent on a per-packet basis. Constant fields can be stored at the receiver to decompress the packet, provided that there is a mechanism to properly keep that information synchronized.

This information on the link is kept synchronized by occasionally sending some uncompressed packets. These uncompressed "reference" packets will convey all the constant data necessary to expand compressed packets that may be sent in the future. The details of when reference packets are sent varies with the protocol and its error characteristics, but, for the most part, reference packets will only be sent only a small fraction of the time.

Since a host may have several simultaneous connections, multiple sets of constants are allowed to coexist in different contexts. The problem is that individual streams of packets that each have constants within their own stream may not share constants. This is similar to the idea of having multiple lines in a memory cache. Thus, to allow multiple sets of constants and hence multiple simultaneous connections efficiently, a small number of isolated contexts are introduced. Normally, we use the same default value of 16 possible contexts that VJ compression uses. In specialized environments, however, it may be beneficial to use a lower or higher maximum– e.g. cellular phones may possibly only have one active connection while PPP servers might have many more. Each packet is labeled with a Context ID to show which set of constants it should use as its context.

Context IDs are managed using a least-recently-used (LRU) algorithm. A context is defined by a set of matching constant fields. To determine a context for an outgoing packet, its fields are checked against the constant fields of each active context. If a match is found, the packet is sent with the matching context; otherwise, an older context ID is reclaimed and used at the cost of sending an uncompressed reference packet to change the context's definition. Since the number of contexts is preallocated, there is no need to free context IDs before they get reused. This approach works regardless of whether the protocol is connection-oriented or connectionless.

Given the classifications for the fields in a protocol, the compressed packet format is easy to determine. Essentially, fields that need to be sent are packed in order by their category. The general layout is similar to how VJ TCP packets are formatted, with a bit-field, the delta fields, and the random fields, since that is a reasonable way to place the data. However, the goal is not to mimic VJ TCP or any other format exactly[1], but rather to generate an efficient compressed packet format for any set of specified fields. Since this platform generates both the compression and uncompression code, there is a fair amount of latitude for producing an efficient encoding.

As illustrated in Figure 2, the first portion in a typical compressed header is the Context ID. The next portion contains a bit-field, mostly for flagging when a delta value has changed. In case the flags are not byte-aligned, our implementation pads the bit field to make it byte-aligned. As an optimization, the implementation attempts to fill any unused space with any existing 1-bit long random fields. Otherwise, the bits are assigned sequentially for each delta.

Following the flags are the delta values–the change between the fields in two consecutive packets. These values are placed in the order of the flagged bit fields. Finally, the last bytes of the compressed header before the payload are the random values. The order of the random values, and those of the delta bit fields, are predetermined and do not change between packets.

One further step would have been to allow the user to further control what the compressed packet would look like. For example, one

[1] If the user wants VJ TCP exactly, link layers such as PPP typically employ a compression negotiation protocol for interoperability by which VJ TCP could be used for TCP packets, generated RTP could be used for RTP packets, generated UDP could be used for non-RTP UDP packets, etc.

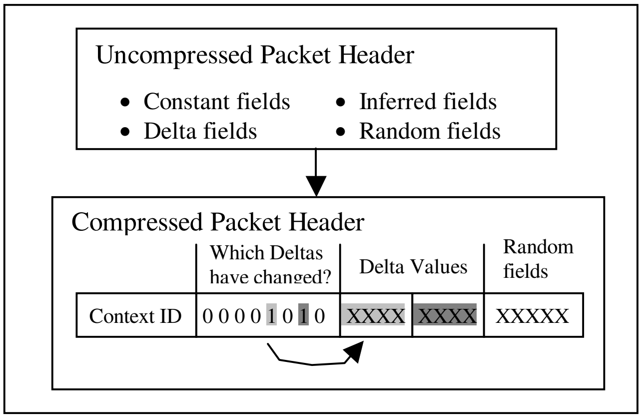

Figure 2: Layout of a typical compressed header, including Context ID, bit-field for the deltas, values for the selected deltas, and random values.

could assign specific bits or combinations of bits[2] to have predetermined meanings. However, at this point, we are not certain about the general usefulness of such a feature.

4 Specification language

The protocol description language is designed to be a simple method for describing the characteristics of a protocol's header fields. This consists of describing individual fields as well as higher-level rules and actions. The language itself is simple, allowing for a parser implementation as a set of Perl programs [36]. We emphasize that the designer of an arbitrary higher-layer protocol only needs to write this description; everything else is automatically taken care of.

To illustrate the semantics of our high-level description language we refer to the specification code in Figure 3, which is used to compress IP headers. Some other protocols are included in the appendices.

The set of lines at the beginning includes the packet identification codes to be used by the link layer, PPP in this case, to tag the compressed and uncompressed packets that the compressor sends. It simply serves as a unique type to identify this compression scheme.

4.1 Fields

The next group of lines beginning with PField characterizes the header fields. The syntax for field descriptions is:

```
PField <name>, fsize=<size in bits>,
   ptype=<NOCHANGE, DELTA, INFERRED,
   RANDOM>;
```

First, the user must describe the fields in the order that they ap-

```
class IPCompress {
        NiceName Generated_IP_Compressor;
        Compressed_ID 0x0081;
        UnCompressed_ID 0x0083;

        PField protover, fsize=4, ptype=NOCHANGE;
        PField hdrlen, fsize=4, ptype=NOCHANGE;
        PField tos, fsize=8, ptype=NOCHANGE;
        PField totlen, fsize=16, ptype=INFERRED,
              formula=[length];
        PField packetid, fsize=16, ptype=DELTA,
              encoding=VARONETHREE,
              negatives=DISALLOWED;
        PField fragments, fsize=16, ptype=NOCHANGE;
        PField ttl, fsize=8, ptype=NOCHANGE;
        PField prot, fsize=8, ptype=NOCHANGE;
        PField checksum, fsize= 16, ptype=RANDOM;
        PField sourceIP, fsize=32, ptype=NOCHANGE;
        PField destIP, fsize=32, ptype=NOCHANGE;

        PRule SendAsIP, ruletext=[protover]!=4; # IPv4 only

        StateVar expireTime, type=int, initvalue=0;
        PRule SendReference, ruletext=[curTime]>{expireTime};
        PAction WhenSendReference, actiontext=
              {expireTime}=[curTime]+5*[ticksPerSecond];
        PAction WhenSendReference, actiontext=
              PrintDebugMessage("Sent reference packet\n");
        PComment WhenSendCompressed, actiontext=
              PrintDebugMessage("Sent compressed packet\n");
}
```

Figure 3: Major portion of a specification for a simple IP compressor

pear in the actual header. Thus, in the 20-byte IP header, the protocol version field comes first and the destination address comes last. Next, when identifying each particular field the user can arbitrarily choose the name of the field. Then the size of the header field is specified in bits, so for example sourceIP, which is the source IP address field in IP, is marked with fsize=32 for thirty-two bits. Any combination of bit sizes is allowed, but the resulting implementation is more efficient if all the fields are byte-aligned. Following the field size is the field type, which we described earlier. Because the Source IP address does not change between packets, we mark it as ptype=NOCHANGE to indicate a Constant field type.

The label NOCHANGE implies that for a given "context" or connection, the field's value does not change. With this in mind, a packet needing to be compressed is determined to be in a given context if all its NOCHANGE fields match the NOCHANGE fields of the context. If the packet matches none of the available contexts, an LRU algorithm finds an older context to reuse.

Random fields, such as the checksum, need no other information since by definition they are sent verbatim every time.

The two additional field types are Inferred and Delta. Inferred fields are not sent in a compressed header, but the compression code must know how to recalculate the field. Therefore when an Inferred field is specified, the user must include a formula, which is then be translated and integrated into the resulting C code. The syntax is as fol-

[2] For instance, in VJ TCP, $(URG + WIN + SEQ)$ together signify echoed terminal traffic, where the SEQ and ACK are both incremented by the last packet size.

lows:

```
PField <name>, fsize=<size in bits>,
    ptype=INFERRED, formula=<reconstruction
    formula>
```

The reconstruction formula above can use arbitrary expressions, including values from the header and special keywords. These substituted values and keywords must be enclosed in brackets (e.g. [protover]), after which our generation tool uses regular-expression processing to translate these into C expressions. For the totlen field in the example description, the value [length] is a special keyword corresponding to the frame length. Beyond that, the values of any field can be used, so [totlen-hdrlen*4] would yield the value of the totlen field minus four times the hdrlen field. There are a number of other places in the file using this bracket notation, which adds power to the specification language.

Some other features have been added to the reconstruction formula language to suit different needs. Using the % sign in the brackets, such as [%UrgPointer], indicates whether or not the field has changed, which is useful in a TCP/IP description. Various connection state variables can be kept after using the StateVar directive. These state variables, referenced in braces, include {expireTime} in Figure 3, and can be used to periodically refresh state in a connectionless protocol.

For safety, the generated compressor checks whether the inferred formula does in fact produce the correct results before sending the packet. In the event that it does not, an uncompressed reference packet is sent. This feature is also useful for encoding fields which rarely change from a given value–a field can be set with formula=x, and in the rare case where that is not true, a full packet is sent.

Deltas are the final field types in the language. These values are supposed to change by small amounts, or perhaps not at all. Typically, the difference in these fields between packets can be sent in less space than the whole fields themselves (e.g. a 32-bit sequence number that increments by one segment size every packet), so that is the general strategy. The complete syntax follows:

```
PField <name>, fsize=<size in bits>,
    ptype=DELTA, encoding=<encoding method>,
    negatives=<[DIS]ALLOWED>
```

Optimizations on this can be made. Since some fields seldom change between packets, a bit-field is employed to prevent many changes of 0 from being sent. And, in order to obtain maximum flexibility, we also support a number of different *encodings* for the differences.

Some applications may benefit from using a fixed-length encoding scheme, knowing that the difference between fields almost always fits in perhaps 8, 16, or 32 bits. In the case where the difference does not fit, the packet needs to be sent uncompressed. Otherwise, variable length encoding schemes may work well. The VARONETHREE encoding, similar to the method for encoding differences in VJ TCP, sends 8-bit differences (1-255) in one octet and 16-bit differences (256-65535) in three octets. It uses an initial zero to signal the latter format. Variable length encoding works fairly well for many fields in TCP/IP, but if this is not the case in others, the encoding of a Delta field can be appropriately changed. Many other possible encoding variations can be easily added to the language, such as incorporating a "default change amount" for fields that often change by a certain fixed size or adding a mechanism for an adaptive encoding algorithm that learns how a field changes.

Other options for deltas include disallowing negative delta values, which is useful for detecting a TCP retransmission; if an earlier packet is resent, that causes a negative change in sequence number, and it may signify than an error occured. This aids in error recovery.

4.2 Rules and actions

To further augment the language, we have provided the ability to verify rules and execute actions in critical places. The basic syntax for a rule looks as follows:

```
PRule <Send[AsIP,Reference]>,
    ruletext=<formula to trigger rule>;
```

The rules used in the example description include additional commands for when to pass the packet through as normal IP (SendAsIP) and for when to send the packet uncompressed (SendReference). These rules cause the *ruletext* to be evaluated along with the rules that are naturally in the system.

In the SendAsIP example, the *ruletext* [protover]!=4 triggers the packet not to be compressed if the IP version is not 4. Thus, IPv6 packets would be passed through this specific compressor without processing. The SendReference rule in the example is used to periodically refresh the compression state. In this case, an uncompressed packet is sent every 5 seconds to help keep the context state from errors.

Actions are statements executed at critical locations, and they are formatted as shown below:

```
PAction <whichaction>,
    actiontext=<action to execute>;
```

In the example IP code in Figure 3, an action is used to update a state variable every time an uncompressed packet is sent. In particular, when the WhenSendReference condition is met, its actiontext = {expireTime}=[curTime]+5*[ticksPerSecond] action is executed, which updates a state variable name expireTime, defined elsewhere in the description, with a value derived from two special-purpose quantities, curTime and ticksPerSecond.

Other potential uses of actions include printing debugging messages to the console when certain conditions are met. In addition to the WhenSendReference condition used in this example, other conditions include WhenSendAsIP and WhenSendCompressed, which encompasses the three possible output packet types that this compressor can produce.

5 Error handling

A significant challenge with sending data over any medium, particularly wireless, is dealing with losses. While a corrupt packet in a regular transmission situation can cause that packet to be lost, the main problem is that a corrupt packet with header compression can cause problems for future packets due to maintained state. For example, in the case of VJ header compression, one corrupted packet header will cause all future packets to be corrupted until an uncompressed reference packet is sent, which will generally not happen until a retransmission. This effectively disables TCP's *fast retransmit* algorithm and at least a transmission window worth of data is dropped after the decompressor state is desynchronized from the compressor.

Since the focus of our work is automated header compression generation, we confined the scope of error handling to match that of standard header compressors. Nevertheless, we are confident that the specification language can be extended to convey information needed to support more sophisticated error recovery algorithms. For now, however, we identify two areas where errors may affect future packets: delta fields and context IDs. Deltas, including their bit-fields and values, are susceptible since only inter-packet differences are represented. With context IDs, the number is mutated, so both the intended and unintended contexts become desynchronized. It would be possible to extend the tools to tag the most important fields, such as the context ID, for forward error correction, as long as this does not become excessively expensive in space or speed.

For connection-oriented protocols such as TCP/IP, uncompressed reference packets need to be sent at the beginning of a given context and whenever a corruption occurs, which can be signaled by TCP retransmits. The method used in VJ TCP involves using hints from the transport layer by noting that whenever a TCP retransmission takes place, the requested sequence number will be less than the previous sequence number. This will cause a negative "delta" for the sequence number. The same is true for the ACK numbers on the ACK path. To implement this, the header description language provides the ability on each delta to disallow negative delta values and instead cause packets to be retransmitted. Disallowing the negative fields yields the same general error characteristics as the VJ algorithm. Since the higher layer does the retransmission anyway, we leave detecting and recovering from these errors to the transport layer.

In connectionless protocols, such as those based on UDP, there is usually no feedback that can be deduced from the transport layer. Rather, most approaches make some use of the principle of soft state [2, 27], which can keep the appropriate state updated by periodically sending uncompressed reference packets over the link. In addition, the number of state changes is reduced, with the concept of "*generation numbers*" from Degermark's work on UDP/IP to identify the state change to which a packet belongs [5]. Some of this can be seen in the description of the IP header in Figure 3.

There is some debate over the merits of explicit retransmission requests. The introduction of retransmission makes what could have been a purely simplex transmission into a full duplex conversation and can cause an implosion problem over multicast. The merits of either side of this debate can be seen in reading Jacobson and Casner [1], who advocate retransmission in compressed RTP, and in reading Degermark [5, 19], who advocates a pure simplex model and repair attempts on damaged packet headers. In our implementation, we are agnostic toward either stance, although the code reflects the simplex model. Rather, the error handling characteristics of the tool can be extended as the need arises, perhaps in a protocol-specific manner.

5.1 Analysis

We now analyze the throughput of a TCP connection using header compression, deriving the conditions under which compression is worthwhile. The key to the analysis is to observe that if there are no losses across the bandwidth-constrained link, header compression *always* has a marginal utility, under the reasonable assumption that the computational costs of header compression are negligible.[3] If the loss rate is higher than some threshold, then header compression has the potential to degrade throughput because the loss of one or more packets in a stream desynchronizes the state of the decompressor relative to the compressor, and requires one or more reference packets for recovery. Observe that all the packets sent in the interim are unrecoverable, because they decompress to a message with a bad end-to-end checksum and are therefore discarded by the higher-layer protocol.

Suppose h is the number of bits saved on average due to header compression when no losses occur, b the payload size in bits, and r the transmission bandwidth of the constrained link in bits/second. Let T_{sync} be the average time for state resynchronization to occur after a loss (for TCP, T_{sync} would be on the order of a retransmission timeout, RTO), l the packet loss rate across the constrained link, and RTT the average round-trip time, which is the time to recover a packet using a fast retransmission triggered by three duplicate acknowledgments [33]. We assume for tractability that link packet losses are independently distributed and that a single loss causes desynchronization. Then, when header compression is enabled, single packet losses in effect become burst losses of size $\frac{r}{b} \times RTT$ packets, distributed at the same loss rate as the original case.

Consider the average time for a payload of size b to be transmitted when no header compression is done. With probability $1 - l$, it takes time $\frac{(b+h)}{r}$, while with probability $l(1 - l)$, it takes time $RTT + \frac{(b+h)}{r}$, since the loss of a packet followed by a retransmission occurs in a time duration on the order of RTT, and $\frac{(b+h)}{r}$ is the time for the packet to get successfully get through the second time round. A third term equal to $l^2(1-l)\{\frac{(b+h)}{r} + RTT + RTO\}$ accounts for the case when the original transmission and a retransmission are lost, followed by a successul one after a TCP timeout. We ignore higher-order terms in l, as they occur in the header-compressed case as well, and are not significant when l is relatively small.

[3] Given a relatively low-bandwidth link. Compression times in our implementation are generally on order of 5-25 μs, dominated by a packet memory copy. This is 1-3 bits on a 115kbs link. If it becomes more significant, it can simply be subtracted from the header savings h in the analysis.

Simplifying this expression leads to the following expression for T_{nocomp}:

$$T_{nocomp} = \frac{(b+h)}{r} + l \times RTT + l^2 \times RTO + O(l^3) \quad (1)$$

When header compression is implemented, a single loss requires time T_{sync} to recover, but the no-loss case only takes time $\frac{b}{r}$. This leads to the following expression for $T_{withcomp}$:

$$T_{withcomp} = \frac{b}{r} + l \times T_{sync} + l^2 \times T_{sync} + O(l^3) \quad (2)$$

This T_{sync} factor includes an RTO, but after that, we need to account for the required retransmission. The TCP window consists of up to $\frac{r \times RTT}{b}$ packets, and the connection resumes in slow start [15]. We assume that $ssthresh$ is cut in half, with half of the packets being transmitted using slow start. The number of round trips[4] to reach that point is $log_2(\frac{r \times RTT}{b})$. At this point, the remaining packets can be transmitted in one final round trip since the window size is equal to the amount of remaining data:

$$T_{sync} \approx RTO + RTT(log_2(\frac{r \times RTT}{b})) \quad (3)$$

Thus, we conclude that $T_{withcomp} < T_{nocomp}$, provided that $l < \frac{h}{r \times (T_{sync} - RTT)}$, which yields:

$$l < \frac{h}{r \times [RTO + RTT \times log_2(\frac{r \times RTT}{b})]} \quad (4)$$

For a typical telnet-like TCP application over a 28.8 Kbps link with an RTO of 1.5 s and an RTT of 200 ms, header compression is a gain when the packet loss rate is less than about 0.3%. Lower transfer rates allow it to be a gain at higher loss rates. Figure 4 shows how beneficial it is for different payload sizes to be compressed with these same parameters as a function of the bit error rate.

For an unreliable protocol such as RTP [29] where corrupted data is dropped rather than retransmitted, the application observes a higher *burst* error rate. Given an average synchronization time after a loss of $T_{refresh}$, each single packet loss will cause a burst loss of $T_{refresh} \times r$ packets, where r is the stream's packet transmission rate. This in effect also magnifies the compressed packet loss rate l_{comp} to $l \times (T_{refresh} \times r)$, assuming that errors on input loss rate l are uniformly distributed.

The value of $T_{refresh}$ here depends on the resynchronization mechanism. A simple periodic refreshing mechanism has the potential to produce large bursts, whereas an explicit notification scheme may limit this burst size on the order of an RTT. A packet repair mechanism such as the "twice" algorithm [5] decreases the loss rate l by avoiding the need to resynchronize in some cases.

How this effects the end-to-end protocol and target application depends on the protocol's tolerance for error. Any application using

[4]We assume no delayed ACKs. The analysis for delayed ACKs is similar.

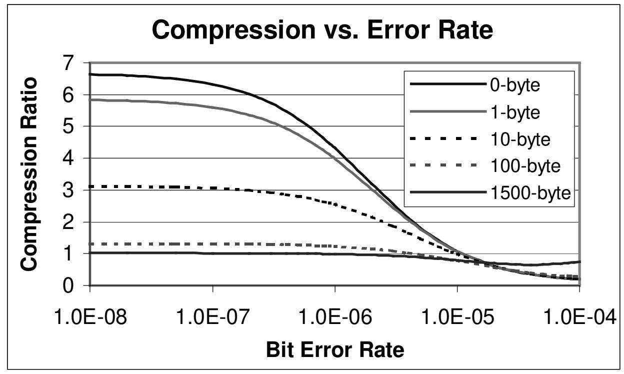

Figure 4: Header compression ratio for different payload sizes as the bit error rate is varied, assuming a 28.8Kbps link, an RTT of 200ms, and an RTO of 1.5s. When the ratio falls below one, header compression is not advantageous.

an unreliable protocol should be expected to operate at a "normal" quality under some error rate, l_{max}. Consequently, if using header compression increases the loss rate beyond what is acceptable to the application, e.g., $l_{comp} > l_{max}$, the compression should be turned off.

6 Implementation

In this section we describe the tools for converting a specification in the description language to a target platform and the approach for using the produced code. The process for deploying header compression in any environment can be thought of in three parts: *writing* the protocol description, which is platform-independent; *generating* the C code from the description; and *deploying* the produced code. Since the header description language was described in Section 3, the latter two parts occupy this section.

6.1 Code generation

The tools for generating the C code were written using Perl scripts for simplicity. We also used a utility called Jeeves [32], a Perl-based tool for aiding the development of code generation programs. The scripts work by parsing the protocol's high-level description and then expanding a C "*template*" file based on the description.

The template file is the code outline that the tools use to translate the header information found in the description file to protocol-dependent C compression code. The template file is essentially C code with Perl generation directives interspersed. Instead of presenting every aspect of the code generation process, we mention some particulars of the methodology. This includes the means by which we implement the protocol fields and an outline of the core functions in the output file. We note that higher layer protocol designers do not have to deal with this template code at all.

6.1.1 Core compression functions

In our implementation, after the tool has evaluated the protocol description and template files, it generates C code for the compression routines and various support files for integration into the Linux kernel. The core of the compressor consists of three routines:

```
int compress(struct connect_data, char*
    input_buffer, int buffersize, char* output_buffer,
    char** real_output, int* output_type);

int uncompress(struct connect_data, char*
    input_buffer, int buffersize);

int remember(struct connect_data, char*
    input_buffer, int buffersize);
```

The above are function prototypes for the compression routines that are generated by the tool. Compress() decides whether an outgoing packet can be compressed can send it as one of three types:

- Regular IP (entirely passed through, e.g., a UDP packet through a TCP compressor),
- Compressed packet (deltas + random)
- Uncompressed reference packet (all the data)

On the receiving end, platform support code calls uncompress() when a compressed packet is received and remember() when an uncompressed reference packet arrives. The remember() function stores the constants and last delta values necessary to decompress future packets for a given Context ID. Likewise, uncompress() uses this stored context state to decompress the packets it receives. Since the tool produces both the compression and decompression portions, it is free to make as many optimizations as possible between the sender and receiver to reduce the link bandwidth.

6.1.2 Field support

In the generated C compressor code, one key to the design is creating #define macros to abstract the process of retrieving and setting each header field. This is particularly important since we allow processing of fields at any bit alignment.

To produce these #define macros, we have a large amount of Perl code near the top of the template file to generate appropriate accessor functions given the bit sizes and alignments. The Perl code needs to generate C #define code to correct for network byte order and to work at any bit alignment. But, for the common byte-aligned case, the code is generated as efficiently as possible. Shown below is an example of the generated #define code.

```
#define GEThdrlen(obj) (((obj)[0]>>(0))&15)

#define SEThdrlen(obj,val)
    ((obj)[0])=((((obj)[0])&~15) | ((val&15)<<0))
```

GEThdrlen and SEThdrlen are generated by the tool based on the framework in our template file. Since the header length field is only four bits and because C does not provide a simple way to access bit information, this tool generates the appropriate shifts and masks to access the data.

6.1.3 Platform support code

Before any of these compression functions can be used on incoming data, some support infrastructure needs to be in place. The compression code is generated using a template, though separately from the core compressor code to isolate platform-specific portions. We used the Linux operating system in our implementation, and we used PPP as the underlying link layer because most low-bandwidth links today use PPP framing.

PPP is a protocol to tunnel IP and other protocols, such as IPX [24] and Appletalk [11], over a point-to-point link, and is particularly suited for low bandwidth links. There are two important capabilities in PPP that support these new compression modes, which are packet tagging and its native header compression negotiation protocol.

Each packet sent in PPP is tagged with a specific packet type. This allows the receiver to give incoming packets to the correct processing code–IP, AppleTalk, and VJ TCP compressed packets can be demultiplexed to the proper receiving routines. These packet tags, standardized by the IETF, are 16-bit quantities [31], yet only a small number of them[5] are used in practice. It is foreseeable that the IETF could allocate an "experimental" region for these tags. In any case, we allocate two packet tags for every new header compressor. This includes a tag to identify Compressed packets and a tag to identify Uncompressed Reference packets. Packets that are not supported by a given compressor retain their Regular IP tag. At that point, Regular IP packets may in fact be processed in turn by subsequent header compressors until one can successfully handle that packet type.

The IP Control Protocol in PPP (IPCP) is then used to negotiate which packets can be sent within a PPP connection [23]. Among its many options, the IPCP can negotiate for supported header compressor algorithms for the link. This is an extensible protocol already used for negotiating VJ TCP compression. Here, each side communicates the compression protocols which it understands, identified by their packet tags, and proceeds by sending affirming and negating messages until the maximal common set is mutually agreed upon. The platform-specific template file generates this negotiation code for every header compressor specification in the description.

6.2 Platform deployment

The tools eventually generate fifteen platform-specific files in the Linux implementation for the compression infrastructure, in addition to a C and header file produced for each generated header compressor type. These platform files include:

- Kernel PPP code, including the compression/decompression routines and modifications to allow the PPP code to call these new routines.
- Modifications to the PPP daemon, so that it might negotiate the use of the generated algorithms with other PPP daemons using the IPCP.

[5] Less than a dozen (out of a namespace represented by a 16-bit field) in our Linux PPP v2.3.5 implementation.

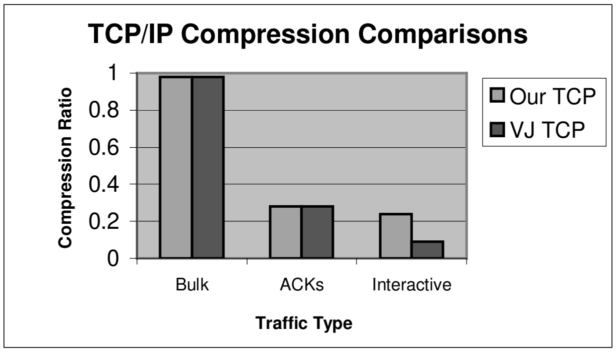

Figure 5: Comparing compression ratios of different classes of traffic between generated TCP code and Van Jacobson TCP compression.

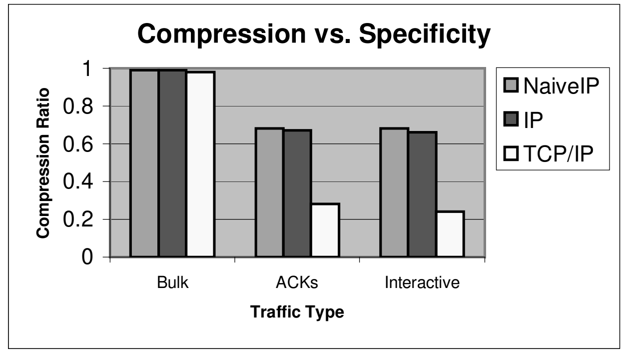

Figure 6: An illustration of more specific characterization of a protocol (Naive IP vs. IP vs. full TCP/IP) allowing better compression.

- An enhanced PPPStats program to generate extra statistics and to aid in debugging.

In practice, compilation and deployment is not very difficult–the generating scripts produce two tar files, one which should be unpacked in the kernel source directory and the other which should be unpacked in the PPP source directory. At this point, the kernel does need to be rebuilt by invoking a Makefile, but otherwise the deployment process is fairly straightforward. With extra scripts, this could be automated to generate appropriate files for many platforms, but we have found the current deployment process to be convenient enough.

7 Results

We have a working implementation of the system, including descriptions for a number of protocols such as TCP/IP, UDP, and RTP. In addition, the header compression language supports features such as per-context state variables and using them in formulae, as well as support for passing variable length fields such as IP options.

The experiments for this section were done using a laptop computer connected to a desktop host via a 115Kbps null modem cable. Both machines are Pentium II systems, running the RedHat 5.2 Linux distribution, including kernel version 2.0.36 and PPP version 2.3.5. We made further modifications in the kernel to simulate drop rates.

7.1 Performance

One of our major goals was to compare the generated compression protocols with hand-produced equivalents, both in terms of space and speed efficiency. Despite the different protocols that could be generated, the best available baseline for comparison is VJ TCP, which has been deployed widely in part due to its space and speed efficiency.

The three classes of traffic analyzed for the TCP/IP tests were Bulk, ACK, and simulated Interactive traffic. The Bulk data test included transferring a 3MB file using scp over the 115kbps link. The MTU was 1536 bytes. The ACK traffic corresponds to the bulk data ACKs from the reverse link for this test. Interactive traffic was simulated with a Java program that sent a text file to the destination echo port with a random inter-character time distributed between 0 and 250 ms. The comparison graphs display the ratio between the total number of bytes sent after using the compressor, including those sent as uncompressed and reference packets, to the number of bytes input to the compressor.

From the comparison graph in Figure 5 between our generated TCP/IP compressor implementation and the VJ implementation, we make several observations. For bulk traffic and ACK traffic, the results are roughly even. In the case of bulk traffic, there were no significant gains from saving 30 bytes on a 1500 byte packet. ACK traffic, since it has no payload, experiences much better compression properties, which might also speed TCP/IP window size convergence. Interactive traffic, which primarily consists of 41-byte packets compared to 40-byte ACK packets, compresses better than ACK traffic in both cases. This is partially due to smaller inter-packet deltas that can be encoded in 1 byte rather than 3 bytes. But, the performance for the VJ algorithm is still noticeably better. The better performance stems from some special cases the VJ algorithm handles, such as echo traffic mode. This echo mode is signified by special-case bit combinations and signifies that the ACK and Sequence numbers both increment by the last packet size. From this observation, we learn that the benefit from handling this special case is in fact significant, and that adding these for our TCP compressor would be a good idea.

One hypothesis that we wanted to quantify was whether giving more specific information about a protocol enables a more efficient compressor to be written. To this end, we generated three compressors, giving them different amounts of information about the TCP/IP protocol. This included a Naive IP compressor, which treats every field as a delta; an IP-only compressor; and our TCP/IP compressor from the last example. From the tests shown in the Figure 6, we find that extra specificity helps, but in this case, defining a larger part of header (e.g. IP vs. TCP/IP) gave a better incremental improvement than defining an existing portion better (e.g. Naive IP vs. IP). The difference between the Naive IP and regular IP compressor was much smaller than expected, which indicates that a naive compressor is not a particularly poor compressor. Most of the improvement came after adding the 20-byte TCP header to the

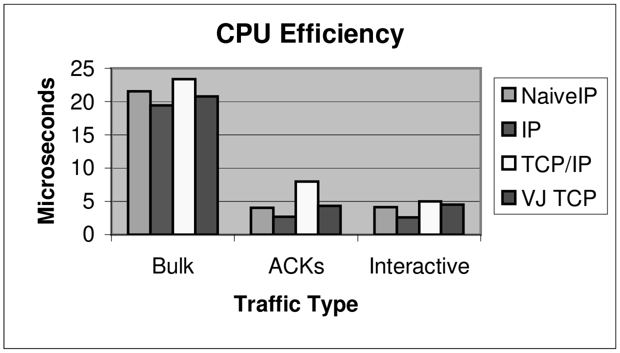

Figure 7: Average CPU time in protocol compression code by classes of traffic for three generated protocols, then for Van Jacobson compression.

analysis. This indicates that it may be fruitful to continue work to higher-level layers, research that our platform effectively enables.

To test CPU efficiency, we measured the time used by the compression code with CPU cycle counters. We tested the three different compressors outlined in the specificity analysis and compared the results to the results of VJ TCP/IP. Overall, as the graph in Figure 7 shows, the generated code does incur a performance penalty, particularly with the full-blown TCP/IP example, but it is relatively small. Much of the inefficiency comes from the simplifying abstractions and not being able to do comparisons and assignments in the most efficient manner. One factor that buffered the differences across the board in our specific platform is that the compressor needs to do a memory copy of the payload to the end of the compressed packet being constructed. This dominates, particularly in the bulk case.

One interesting factor we observed was that given a description of a protocol in an RFC (e.g. RTP), we could often implement a rough characterization of it using the generation tools in much less time than it took to compile the resulting code and test the protocol. There was some trial and error in refining the various protocols descriptions, as an incorrect characterization could cause all packets to be sent as uncompressed. One could also further refine the generated C code if desired, but what we have seems to be a promising way to compress arbitrary end-to-end protocols.

8 Discussion

After finding that specificity does significantly increase the protocol's space efficiency, particularly as the specification spans more protocol layers, an obvious next step is to look at other higher-level protocol headers. Looking at usage trends on the Internet, we proceeded to consider string-based protocols such as HTTP [9].

The specification language as described works well for binary formats where most of the data fields are fixed in length. We have experimented with support for some variable length fields as would be needed to work with IP options, but one step beyond this is support for string-based protocols, particularly HTTP. Before embarking on implementing string-based primitives, we first ran some tests on HTTP trace data to determine if the task would be worthwhile.

Using dialup Web trace data from UC Berkeley [10], we found that even with making generous assumptions for the size the the HTTP request header, only 6.3% of the data packets transmitted consist of HTTP request and response headers. Seeking to minimize the effects of a few outlying large files, we found that restricting the statistics to downloads under 250kB, 50kB, and 10kB yielded total header bandwidth of 9.3%, 11.1%, and 19.9%, respectively. Even if many web downloads are under 10kB, we did not feel that an improvement on the order of 20% would be worth putting the equivalent of an HTTP parser in the operating system kernel, even one that was automatically generated.

For this reason, while we recognize that there could be some observable gains from a compressor that simply tokenized string-based protocols and kept minimal state, we decided that this type of compression for the most part would not be worth pursuing immediately in the context of HTTP.

In addition, some further work on error handling and adding support for either error reconstruction algorithms such as the "twice" algorithm [5] or explicit retransmission requests [19] is one more area where work can be done. More work as well on efficiently packing the data into a compressed packet format, as well as providing special flag combinations to exploit for the common case, may also be helpful.

Integration and analysis with IP Security [22] is another area for future work. While very little of an Encapsulating Security Payload [21] packet header is compressible, packets using only the Authentication Header [20] can be compressed since the fields are unencrypted and the header can be fully reconstructed for the receiver authentication layer to process.

We believe that our work has demonstrated the effectiveness of a unified header compression framework and that it is relatively easy for higher-layer protocol designers to specify compression directives in a simple language. More work in this area would also include interfacing to more operating systems and link layers. In our own experience, a disproportionate amount of the effort was spent on Linux hacking and getting the platform to work correctly. We suspect from the experience that the amount of necessary background kernel effort required for any header compression does in fact contribute to hindering its widespread implementation and deployment.

9 Conclusion

We conclude that it is in fact possible to automatically generate header compression algorithms both quickly and efficiently from a high-level description of the protocol. Furthermore, we find that the generated code can behave roughly comparably to hand-designed algorithms and code. Our tool is flexible enough to handle several available protocols, and it can be expanded upon in the future to support emerging protocols and applications. We therefore believe that it is the right direction given the increasing proliferation of application-level protocols.

The source code and examples for the tool are available at `http://nms.lcs.mit.edu/software/headercompress/`.

References

[1] CASNER, S. AND JACOBSON, V. *Compression IP/UDP/RTP Headers for Low-Speed Serial Links*. Internet Engineering Task Force, Feb. 1999. RFC-2508.

[2] CLARK, D. The Design Philosophy of the DARPA Internet Protocols. In *Proc. ACM SIGCOMM* (Aug. 1988).

[3] CORPORATION, M. Windows Media Technologies. `http://microsoft.com/ntserver/mediaserv/`, 2000.

[4] DEERING, S., AND HINDEN, R. *Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, Dec 1995. RFC-1883.

[5] DEGERMARK, M., ENGAN, M., NORDGREN, B., AND PINK, S. Low-loss TCP/IP Header Compression for Wireless Networks. In *Proc. MOBICOM 1996 (Rye, NY)* (Nov. 1996).

[6] DEGERMARK, M. NORDGREN, B. PINK, S. *IP Header Compression*. Internet Engineering Task Force, Feb. 1999. RFC-2507.

[7] ENGAN, M. AND CASNER, S. AND BORMANN, C. *IP Compression over PPP*. Internet Engineering Task Force, Feb. 1999. RFC-2509.

[8] FARBER, D. AND DELP, G. AND CONTE, T. *A Thinwire Protocol for connecting personal office computers to the Internet*. Internet Engineering Task Force, Sept. 1984. RFC-914.

[9] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., AND BERNERS-LEE, T. *Hypertext Transfer Protocol—HTTP/1.1*. Internet Engineering Task Force, Jan 1997. RFC-2068.

[10] GRIBBLE, S. D. UC Berkeley Home IP Web Traces. `http://www.acm.org/sigcomm/ITA/`, July 1997.

[11] GURSHARAN, S., ANDREWS, R., AND OPPENHEIMER, A. *Inside Appletalk*. Addison Wesley, 1990.

[12] HANDLEY, M. AND JACOBSON, V. *SDP: Session Description Protocol*. Internet Engineering Task Force, Apr. 1998. RFC-2327.

[13] HART, R. Linux PPP HOWTO. `http://www.linuxdoc.org/`, Mar. 1997.

[14] HOWARD, J. An Overview of the Andrew File System. In *Proc. 1988 USENIX Winter Conference (Dallas, TX)* (Feb. 1988), pp. 23–26.

[15] JACOBSON, V. Congestion Avoidance and Control. In *Proc. ACM SIGCOMM* (Aug 1988).

[16] JACOBSON AND BRADEN AND BORMAN. *TCP Extensions for High Performance*. Internet Engineering Task Force, May 1992. RFC-1323.

[17] JACOBSON, V. *Compressing TCP/IP Headers for Low-Speed Serial Links*. Internet Engineering Task Force, Feb. 1990. RFC-1144.

[18] JOHNSON, D. Scalable support for transparent mobile host internetworking. In *Mobile Computing*, T. Imielinski and H. Korth, Eds. Kluwer Academic Publishers, 1996, ch. 3, pp. 103–128.

[19] JONSSON, L., DEGERMARK, M., HANNU, H., AND SVANBRO, K. RObust Checksum-based header COmpression (ROCCO). Internet Draft draft-jonsson-robust-hc-03.txt, Jan. 2000. Work in progress.

[20] KENT, S. AND ATKINSON, R. *IP Authentication Header*. Internet Engineering Task Force, Nov. 1998. RFC-2402.

[21] KENT, S. AND ATKINSON, R. *IP Encapsulating Security Payload (ESP)*. Internet Engineering Task Force, Nov. 1998. RFC-2406.

[22] KENT, S. AND ATKINSON, R. *Security Architecture for the Internet Protocol*. Internet Engineering Task Force, Nov. 1998. RFC-2401.

[23] MCGREGOR, G. *The PPP Internet Protocol Control Protocol (IPCP)*. Internet Engineering Task Force, May 1992. RFC-1332.

[24] NOVELL, I. Advanced NetWare V2.1 Internetwork Packet Exchange Protocol (IPX) with Asynchronous Event Scheduler (AES), Oct. 1986.

[25] NOWICKI, B. *NFS: Network File System Protocol Specification*. Internet Engineering Task Force, Mar 1989. RFC-1094.

[26] PARTRIDGE, C. AND HINDEN, R. *Version 2 of the Reliable Data Protocol (RDP)*. Internet Engineering Task Force, Apr. 1990. RFC-1151.

[27] RAMAN, S., AND MCCANNE, S. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proc. ACM SIGCOMM* (Sept. 1999), pp. 15–25.

[28] REALNETWORKS. Real Player. `http://www.real.com/player/`, 1999.

[29] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. *RTP: A Transport Protocol for Real-Time Applications*. Internet Engineering Task Force, Jan 1996. RFC-1889.

[30] SCHULZRINNE, H. AND RAO, A. AND LANPHIER, R. *Real Time Streaming Protocol (RTSP)*. Internet Engineering Task Force, Apr. 1998. RFC-2326.

[31] SIMPSON, W. *The Point-to-Point Protocol*. Internet Engineering Task Force, July 1994. RFC-1661.

[32] SRINIVASAN, S. Jeeves - A Configurable Application Generator. Berkeley, CA, 1995.

[33] STEVENS, W. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. Internet Engineering Task Force, Jan. 1997. RFC-2001.

[34] SUN MICROSYSTEMS. *RPC: Remote Procedure Call Protocol Specification Version 2*. Internet Engineering Task Force, Jun 1988. RFC-1057.

[35] SUN MICROSYSTEMS. Java Remote Method Invocation (RMI). `http://java.sun.com/products/jdk/rmi/`, 1997.

[36] WALL, L., CHRISTIANSEN, T., SCHWARTZ, R., AND POTTER, S. *Programming Perl*. O'Reilly & Associates, Oct. 1996.

A Other Protocol Descriptions

TCP.sc - a description of the TCP/IP protocol

```
class TCP {
    NiceName Generated_TCP_Lookalike;
    Compressed_ID 0x0085;
    UnCompressed_ID 0x0087;
    Parameter maxslots, ipcp_type = u_char,
        uput=PUTCHAR, uget=GETCHAR, size=1,
        ioctlname=PPPIOCSMAXGEN0081SLOT,
        pmin=2, pmax=16, pdefault=16;
    PField protover, fsize=4, ptype=NOCHANGE;
    PField hdrlen, fsize=4, ptype=NOCHANGE;
    PField tos, fsize=8, ptype=NOCHANGE;
    PField totlen, fsize=16, ptype=INFERRED,
        formula=[length], checkformula=false;
    PField packetid, fsize=16, ptype=DELTA,
        encoding=TWOBYTE, negatives=DISALLOWED;
    PField fragments, fsize=16, ptype=NOCHANGE;
    PField ttl, fsize=8, ptype=NOCHANGE;
    PField prot, fsize=8, ptype=NOCHANGE;
    PField IPchecksum, fsize= 16, ptype=RANDOM;
    PField sourceIP, fsize=32, ptype=NOCHANGE;
    PField destIP, fsize=32, ptype=NOCHANGE;
    PField TCPsourcePort, fsize=16, ptype=NOCHANGE;
    PField TCPdestPort, fsize=16, ptype=NOCHANGE;
    PField TCPsequenceno, fsize=32, ptype=DELTA,
        encoding=VARONETHREE,
        negatives=DISALLOWED;
    PField TCPACKNo, fsize=32, ptype=DELTA,
        encoding=VARONETHREE,
        negatives=DISALLOWED;
    PField TCPhdrlen, fsize=4, ptype=NOCHANGE;
    PField TCPreserved, fsize=6, ptype=NOCHANGE;
    PField URGflag, fsize=1, ptype=INFERRED,
        formula=[%UrgPointer], checkformula=true;
    PField ACKflag, fsize=1, ptype=RANDOM; PField
        PSHflag, fsize=1, ptype=RANDOM;
    PField RSTflag, fsize=1, ptype=INFERRED,
        formula=0, checkformula=false;
    PField SYNflag, fsize=1, ptype=INFERRED,
        formula=0, checkformula=false;
    PField FINflag, fsize=1, ptype=INFERRED,
        formula=0, checkformula=false;
    PField WindowSize, fsize=16, ptype=DELTA,
        encoding=VARONETHREE,
        negatives=DISALLOWED;
    PField TCPchecksum, fsize=16, ptype=RANDOM;
    PField UrgPointer, fsize=16, ptype=DELTA,
        encoding=VARONETHREE,
        negatives=DISALLOWED;
    PFlag DetailedDebugMessages, status=off;
        # want IP version 4
    PRule SendAsIP, ruletext=[protover]!=4;
        # don't handle IP options here
    PRule SendAsIP, ruletext=[hdrlen]<5;
        # make sure it's TCP (proto#6)
    PRule SendAsIP, ruletext=[prot]!=6;
        # don't send fragments
    Prule SendAsIP, ruletext=[fragments]&0x3fff;
        # don't send SYN/FIN/RST packets
    PRule SendAsIP, ruletext=[SYNflag]!=0;
    PRule SendAsIP, ruletext=[FINflag]!=0;
    PRule SendAsIP, ruletext=[RSTflag]!=0;
    PComment WhenSendAsIP,
        Actiontext=PrintDebugMessage ("Sent as IP\n");
    PComment WhenSendReference,
        actiontext=PrintDebugMessage ("Sent reference
        packet\n");
    PComment WhenSendCompressed,
        actiontext=PrintDebugMessage("Sent compressed
        packet\n");
    PVarField IPOptions, $ruletext=[hdrlen]>5,
        $sizetext=([hdrlen]-5)*4, $maxsize=54,
        $starttext=4*5;
}
```

NaiveIP.sc - a "naive" protocol description based on Thinwire [8] which contains a large bit-field to indicate which of the first 20 header octets have changed.

```
class NaiveIP {
    NiceName Naive_IP;
    Compressed_ID 0x0089;
    UnCompressed_ID 0x008b;
    Parameter maxslots, ipcp_type = u_char,
        uput=PUTCHAR, uget=GETCHAR, size=1,
        ioctlname=PPPIOCSMAXGEN0081SLOT,
        pmin=2, pmax=16, pdefault=16;
    PField fld1, fsize=8, ptype=NOCHANGE;
    PField fld2, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    PField fld3, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    PField fld4, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    PField fld5, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    ... fields 6 – 17 ...
    PField fld18, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    PField fld19, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    PField fld20, fsize=8, ptype=DELTA,
        encoding=ONEBYTE, negatives=ALLOWED;
    PFlag DetailedDebugMessages, status=off;
    StateVar expireTime, type=int, initvalue=0;
    PRule SendReference,
        ruletext=[curTime]>{expireTime};
    PAction WhenSendReference,
        actiontext={expireTime}=[curTime]+
        5*[ticksPerSecond];
}
```