Consistent Overhead Byte Stuffing

Stuart Cheshire and Mary Baker, *Member, IEEE*

***Abstract*—Byte stuffing is a process that encodes a sequence of data bytes that may contain 'illegal' or 'reserved' values, using a potentially longer sequence that contains no occurrences of these values. The extra length is referred to here as the *overhead* of the encoding. To date, byte stuffing algorithms, such as those used by SLIP, PPP and AX.25, have been designed to incur low average overhead, but little effort has been made to minimize their worst-case overhead. However, there are some increasingly popular network devices whose performance is determined more by the worst case than by the average case. For example, the transmission time for ISM-band packet radio transmitters is strictly limited by FCC regulation. To adhere to this regulation, the current practice is to set the maximum packet size artificially low so that no packet, even after worst-case overhead, can exceed the transmission time limit.**

This paper presents a new byte stuffing algorithm, called Consistent Overhead Byte Stuffing (COBS), which tightly bounds the worst-case overhead. It guarantees in the worst case to add no more than one byte in 254 to any packet. For large packets this means that their encoded size is no more than 100.4% of their pre-encoding size. This is much better than the 200% worst-case bound that is common for many byte stuffing algorithms, and is close to the information-theoretic limit of about 100.07%. Furthermore, the COBS algorithm is computationally cheap, and its average overhead is very competitive with that of existing algorithms.

***Index Terms*—Packet, Serial, Transmission, Framing, Byte stuffing**

I. Introduction

THE PURPOSE of byte stuffing is to convert data packets into a form suitable for transmission over a serial medium like a telephone line. When packets are sent over a serial medium there needs to be some way to tell where one packet ends and the next begins, particularly after errors, and this is typically done by using a special reserved value to indicate packet boundaries. Byte stuffing ensures, at the cost of a potential increase in packet size, that this reserved value does not inadvertently appear in the body of any transmitted packet. In general, some *overhead* (additional bytes transmitted over the serial medium) is inevitable if we are to perform byte stuffing without loss of information.

Current byte stuffing algorithms, such as those used by Serial Line IP (SLIP) [RFC1055], the Point-to-Point Protocol (PPP) [RFC1662] and AX.25 (Amateur Packet Radio) [ARRL84], have highly variable per-packet overhead. For example, using conventional PPP byte stuffing, the total overhead (averaged over a large number of packets) is typically 1% or less, but this overhead is not consistent for all packets. Some packets incur no overhead at all, while others incur much more; in principle some could increase in size by as much as 100%. High-level Data Link Control (HDLC) uses a bit stuffing technique [ECMA-40] that has a worst-case expansion of only 20%, but bit-oriented algorithms are often harder to implement efficiently in software than byte-oriented algorithms.

While large variability in overhead may add jitter and unpredictability to network behavior, this problem is not fatal for PPP running over a telephone modem. An IP host [RFC791] using standard PPP encapsulation [RFC1662] need only make its transmit and receive buffers twice as large as the largest IP packet it expects to send or receive. The modem itself is a connection-oriented device and is only concerned with transmitting an unstructured stream of byte values. It is unaware of the concept of packet boundaries, so unpredictability of packet size does not directly affect modem design.

In contrast, new devices are now becoming available, particularly portable wireless devices, that are packet-oriented, not circuit-oriented. Unlike telephone modems these devices are aware of packets as distinct entities and consequently impose some finite limit on the maximum packet size they can support. Channel-hopping packet radios that operate in the unlicensed ISM (Industrial/Scientific/Medical) bands under the FCC Part 15 rules [US94-15] are constrained by a maximum transmission time that they may not exceed. If the overhead of byte stuffing unexpectedly doubles the size of a packet, it could result in a packet that is too large to transmit legally. Using conventional PPP encoding [RFC1662], the only way to be certain that no packets will exceed the legal limit is to set the IP maximum transmission unit (MTU) to half the device's true MTU, despite the fact that it is exceedingly rare to encounter a packet that doubles in size when encoded.

Halving the MTU can significantly degrade the performance seen by the end-user. For example, assuming one 40-byte TCP acknowledgement for every two data packets [RFC1122], a Metricom packet radio with an MTU of 1000 bytes achieves a maximum end-user throughput of 5424 bytes per second [Che96]. Halving the MTU to 500 bytes reduces the maximum end-user throughput to 3622 bytes per second, a reduction of roughly 33%, which is a significant performance penalty.

Although, as seen from experiments presented in this paper, packets that actually double in size rarely occur naturally, it is not acceptable to ignore the possibility of their occurrence. Without a factor-of-two safety margin, the network device would be open to malicious attack through artificially constructed pathological packets. An attacker could use such packets to exploit the device's inability to send and/or receive

Manuscript received July 29, 1998; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Crowcroft. This work was supported in part by Rockwell International Science Center, FX Palo Alto Laboratory, AT&T/Lucent, under NSF Faculty Career Grant CCR-9501799, and by a grant from the Keio Research Institute at SFC, Keio University & the Information-Technology Promotion Agency of Japan, and by an Alfred P. Sloan Faculty Research Fellowship. This paper was presented in part at SIGCOMM '97, Cannes, France, September 1997.

Stuart Cheshire is with Apple Computer, Cupertino, CA 95014 USA.

Mary Baker is with the Computer Science Department, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 1063-6692(99)03826-1

worst-case packets, causing mischief ranging from simple denial of service attacks to the much more serious potential for exploiting array-bounds errors such as the one in the Unix 'finger' dæmon exploited by the infamous Internet Worm of 1989 [RFC1135].

One way to avoid the performance degradation of halving the MTU might be to set the IP MTU closer to the underlying device MTU, and then use some kind of packet fragmentation and reassembly to handle those packets that, when encoded, become too large to send. Packet fragmentation is not without cost though, as Chris Bennett reported in [Ben82] where he wrote, "As well as the global overheads such as endpoint delay, fragmentation generates overheads for other components of the system — notably the intermediate gateways."

One kind of packet fragmentation is link-layer fragmentation ('transparent fragmentation'). While this could work, requiring link-layer software to perform fragmentation and reassembly is a substantial burden to impose on device driver writers. In [Kent87], Kent and Mogul wrote "… transparent fragmentation has many drawbacks … gateway implementations become more complex and require much more buffer memory …" Fragmentation and reassembly also provides more opportunities for software bugs [CA-96.26] and adds protocol overhead, because the link-layer packet headers have to contain additional fields to support the detection and reassembly of fragments at the receiving end.

A second way to use fragmentation would be to avoid adding a new layer of fragmentation and reassembly at the device level, and instead to use IP's own existing fragmentation and reassembly mechanisms, but this also has problems. One problem is that in current networking software implementations IP fragmentation occurs before the packet is handed to the device driver software for transmission. If, after encoding by the device driver, a particular packet becomes too large to send, there is no mechanism for the driver to hand the packet back to IP with a message saying, "Sorry, I failed, can you please refragment this packet into smaller pieces and try again?" Also, some software goes to great lengths to select an optimal packet size. It would be difficult for such software to cope with an IP implementation where the MTU of a device varies for every packet. Finally, it is unclear, if we were to create such a mechanism, how it would handle the IP header's 'Don't Fragment' bit without risking the creation of path MTU discovery [RFC1191] 'black holes'.

All these problems could be avoided if we used a byte stuffing algorithm that did not have such inconsistent behavior. This paper presents a new algorithm, called Consistent Overhead Byte Stuffing (COBS), which can be relied upon to encode all packets efficiently, regardless of their contents. It is computationally cheap, easy to implement in software, and has a worst-case performance bound that is better even than HDLC's bit stuffing scheme. All packets up to 254 bytes in length are encoded with an overhead of exactly one byte. For packets over 254 bytes in length the overhead is at most one byte for every 254 bytes of packet data. The maximum overhead is therefore roughly 0.4% of the packet size, rounded up to a whole number of bytes.

Using Consistent Overhead Byte Stuffing, the IP MTU may be set as high as 99.6% of the underlying device's maximum packet size without fear of any packet inadvertently exceeding that limit. Thus COBS provides better end-user performance than existing mechanisms, because COBS enables a given piece of hardware to send larger IP packet payloads.

All the data stuffing algorithms discussed in this paper, including COBS, have execution times that are O(n) with respect to the size of the data set being stuffed. The constant factor varies depending on the algorithm and the implementation, but all the algorithms execute in linear time.

The remainder of this paper proceeds as follows: The next section describes two widely used methods for performing data stuffing, HDLC bit stuffing and PPP byte stuffing.

Section III describes our new framing protocol, called Consistent Overhead Byte Stuffing (COBS). COBS guarantees that for all packets, no matter what their contents, the overhead will be small. In addition to the basic algorithm, Section III also presents a simple variant called Zero Pair Elimination (COBS/ZPE) which improves on straight COBS's average performance at the expense of a fractionally higher worst-case bound.

Sections IV and V compare the costs of HDLC, conventional PPP, and COBS for encoding the same data packets. In the least favorable case for COBS, network traffic consisting predominantly of small packets, COBS is found to add on average less than 0.5% additional overhead compared to conventional PPP. Although this is a small price for the performance benefit of being able to use much larger packets, it is possible to eliminate even this cost. COBS/ZPE achieves an average overhead lower than PPP's, even for small-packet traffic, without giving up COBS's advantage of also guaranteeing a very low bound on worst-case overhead. Section IV considers the expected overhead from a theoretical point of view, for data consisting of uniformly distributed random eight-bit values. Real network traffic often does not have a uniform distribution of byte values, so Section V presents experimental results comparing COBS with conventional PPP for real network traffic.

Section VI presents our conclusions.

II. CURRENT DATA STUFFING ALGORITHMS

When packet data is sent over any serial medium, a protocol is needed by which to demarcate packet boundaries. This is done by using a special bit-sequence or character value to indicate where the boundaries between packets fall. Data stuffing is the process that transforms the packet data before transmission to eliminate any accidental occurrences of that special framing marker, so that when the receiver detects the marker, it knows, without any ambiguity, that it does indeed indicate a boundary between packets. Some communications protocols use bit stuffing algorithms (which operate on a bit-by-bit basis), and some use byte stuffing algorithms (which operate on a byte at a time).

Both bit stuffing and byte stuffing in general increase the size of the data being sent. The amount of increase depends on the patterns of values that appear in the original data and can vary from no overhead at all to, in the worst case for PPP, doubling the packet size. Bit stuffing and byte stuffing are discussed in more detail below.

HDLC [ECMA-40] uses a bit stuffing scheme. It uses the binary sequence 01111110, called the Flag Sequence, to mark boundaries between packets. To eliminate this pattern from the data, the following transformation is used: whenever the transmitter observes five ones in a row, it inserts a zero immediately following. This eliminates the possibility of six consecutive ones ever occurring inadvertently in the data. The receiver performs the reverse process: After observing five ones in a row, if the next binary digit is a zero it is deleted, and if it is a one then the receiver recognizes it as one of the special framing patterns. This process of inserting extra zeroes ('bit stuffing' or 'zero insertion') increases the transmitted size of the data. In the worse case, for data that consists entirely of binary ones, HDLC framing can add 20% to the transmitted size of the data.

This kind of bit-level insertion and deletion is easy to implement in the hardware of a serial transmitter, but is not easy to implement efficiently in software. Software gets efficiency from working in units of 8, 32, or more, bits at a time, using on-chip registers and wide data buses. Algorithms that are specified in terms of individual bits can be hard to implement efficiently in software because it can be difficult to take good advantage of the processor's ability to operate on bytes or words at a time. For this reason it is more common for software algorithms to use byte stuffing.

PPP uses a byte stuffing scheme [RFC1662]. It operates in terms of bytes instead of bits and uses a byte with value 0x7E (the same as the HDLC Flag Sequence) to mark boundaries between packets. To eliminate this value from the data payload, the following transformation is used: everywhere that 0x7E appears in the data it is replaced with the two-character sequence 0x7D,0x5E. 0x7D is called the Control Escape byte. Everywhere that 0x7D appears in the data it is replaced with the two-character sequence 0x7D,0x5D. The receiver performs the reverse process: whenever it sees the Control Escape value (0x7D) it discards that byte and XORs the following byte with 0x20 to recreate the original input.

On average this byte stuffing does reasonably well, increasing the size of purely random data by a little under 1%, but in the worst case it can double the size of the data being encoded. This is much worse than HDLC's worst case, but PPP byte stuffing has the advantage that it can be implemented reasonably efficiently in software.

PPP's byte stuffing mechanism, in which the offending byte is prefixed with 0x7D and XORed with 0x20, allows multiple byte values to be eliminated from a packet. For example, PPP byte stuffing can facilitate communication over a non-transparent network by eliminating all ASCII control characters (0x00-0x1F) from the transmitted packet data. It is equally possible to do this using a COBS-type algorithm [Che98], but the subject of this paper is the minimal byte stuffing necessary to facilitate reliable unambiguous packet framing, not extensive byte stuffing to compensate for non-transparency of the underlying network.

A more exhaustive treatment of framing and data stuffing can be found in [Che98].

III. Consistent Overhead Byte Stuffing Algorithms

This section begins by describing the data encoding format used by COBS and the procedure for converting packet data to and from that encoded format. It then describes some of the properties and implications of using COBS encoding, and describes a variant of COBS called COBS/ZPE which has slightly different properties. COBS is described in terms of detecting packet boundaries in a stream of bytes, but it can also be used equally well to decode a raw bit-stream, and this section concludes by describing how this is achieved.

A. COBS Syntax and Semantics

COBS performs a reversible transformation on a data packet to eliminate a single byte value from it. Once eliminated from the data, that byte value can then safely be used as the framing marker without risk of ambiguity.

For the description that follows, zero was chosen as the framing value to be eliminated. In practice zero is a good value to choose for a real-world implementation because zeroes are common in binary computer data, and COBS performs marginally better when it has many bytes to eliminate. However, elimination of some value other than zero can be achieved with only trivial changes to the algorithm, such as the addition of a simple post-processing step (like XORing all the output bytes with the value to be eliminated) [Che97] [Che98].

COBS first takes its input data and logically appends a single zero byte. (It is not necessary actually to add this zero byte to the end of the packet in memory; the encoding routine simply has to behave as if the added zero were there.)

COBS then locates all the zero bytes in the packet (including the added one), and divides the packet at these boundaries into one or more *zero-terminated chunks.* Every zero-terminated chunk contains exactly one zero byte, and that zero is always the last byte of the chunk. A chunk may be as short as one byte (i.e. a chunk containing just a solitary zero byte) or as long as an entire packet.

COBS encodes each zero-terminated chunk using one or more variable length *COBS code blocks*. Chunks of 254 bytes or fewer are encoded as a single COBS code block. Chunks longer than 254 bytes are encoded using multiple code blocks, as described later in this section. After a packet's constituent chunks have all been encoded using COBS code blocks, the entire resulting aggregate block of data is completely free of zero bytes, so zeroes can then be placed around the encoded packet to mark clearly where it begins and ends.

A *COBS code block* consists of a single code byte, followed by zero or more data bytes. The number of data bytes is determined by the code byte. Fig. 1 shows some examples of valid COBS code blocks and the corresponding zero-terminated data chunks they represent.

For codes 0x01 to 0xFE, the meaning of each code block is that it represents the sequence of data bytes contained within

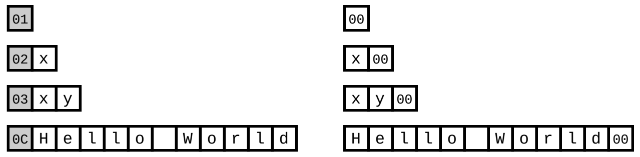

Fig. 1. Example code blocks and the data chunks they represent. Each COBS code block begins with a single code byte (shown shaded), followed by zero or more data bytes.

Code	Followed by	Meaning
0x00	(not applicable)	(not allowed)
0x01	no data bytes	A single zero byte
n	(*n*–1) data bytes	The (*n*–1) data bytes, followed by a single zero
0xFF	254 data bytes	The 254 data bytes, **not** followed by a zero

Table 1. Code values used by consistent overhead byte stuffing.

the code block, *followed by an implicit zero byte*. The zero byte is implicit — it is not actually contained within the sequence of data bytes in the code block.

These code blocks encode data without adding any overhead. Each code block begins with one code byte followed by *n* data bytes, and represents *n* data bytes followed by one trailing zero byte. Thus the code block adds no overhead to the data: a chunk (*n*+1) bytes long is encoded using a code block (1+*n*) bytes long.

These basic codes are suitable for encoding zero-terminated chunks up to 254 bytes in length, but some of the zero-terminated chunks that make up a packet may be longer that that. To cope with these chunks, code 0xFF is defined slightly differently. Code 0xFF represents the sequence of 254 data bytes contained within the code block, *without any implicit zero*. Defining the maximum length code (0xFF) as an exception this way allows us to encode chunks that are too long to be encoded as a single code block. Chunks longer than 254 bytes are encoded using one or more of these special maximum length code blocks, followed by a single normal code block. Each special maximum length code block has no implied trailing zero, but the final (normal) code block does include an implied trailing zero at the end, so this aggregate sequence of code blocks correctly encodes the required long chunk of non-zero data with a single zero byte at the end. Unlike the other code blocks, this maximum length code block does add some overhead to the encoded data. 254 bytes of packet data are encoded using 255 bytes, an overhead of just under 0.4%.

Fig. 2 shows an example of a 680-byte zero-terminated chunk (679 non-zero bytes and one zero) which is encoded using three COBS code blocks. The first two are special maximum-length (254 data bytes, no implied zero) COBS code blocks, and the final code block is a standard 172-byte (171 data bytes and an implied zero) COBS code block. Thus in this case 680 bytes of user data is encoded using 682 bytes, an overhead of just under 0.3%.

The meanings of the various COBS code values are summarized in Table 1.

COBS has the property that the byte value zero is never used as a code byte, nor does it ever appear in the data section of any code block. This means that COBS takes an input consisting of characters in the range [0,0xFF] and produces an output consisting of characters only in the range [1,0xFF]. Having eliminated all zero bytes from the data, a zero byte can now be used unambiguously to mark boundaries between packets. This allows the receiver to synchronize reliably with the beginning of the next packet, even after an error. It also allows new listeners to join a broadcast stream at any time and without fail receive and decode the very next error free packet.

B. COBS Encoding Procedure

The job of the COBS encoder is to translate the raw packet data into a series of COBS code blocks. Breaking the packet into zero-terminated chunks as described above is a useful way to understand the logic of COBS encoding, but in practice it is more efficient to write an encoder that translates from packet data directly to code blocks, without going through an intermediate data format.

The encoding routine searches through the first 254 bytes of the packet looking for the first occurrence of a zero byte. If no zero is found, then a code of 0xFF is output, followed by the 254 non-zero bytes. If a zero is found, then the number of bytes examined, *n*, is output as the code byte, followed by the actual values of the (*n*–1) non-zero bytes up to, but not including, the zero byte. The zero is then skipped and this process is repeated until all the bytes of the packet (including the final zero which is logically appended to every packet) have been encoded. Fig. 3 shows an example of packet encoding.

The implementation of COBS is very simple. The Appendix gives complete C source code listings to perform both COBS encoding and decoding.

There is a small optimization that may be performed in one particular situation. The reason for logically appending a zero to every packet before encoding is that all code blocks except code 0xFF represent a block of data that ends with an implied zero. If the data to be encoded did not actually end with a zero there might be no valid COBS encoding of that data. Adding a zero to the end of every packet circumvents this problem.

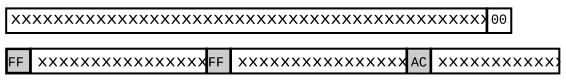
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00
FF xxxxxxxxxxxxxxxx FF xxxxxxxxxxxxxxxx AC xxxxxxxxxxxx

Fig. 2. Encoding oversized chunks.

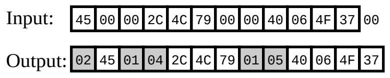
Input:
45 00 00 2C 4C 79 00 00 40 06 4F 37 00
Output:
02 45 01 04 2C 4C 79 01 05 40 06 4F 37

Fig. 3. Example COBS encoding showing input (with a phantom zero logically appended) and corresponding zero-free output, with code bytes shown shaded for clarity.

However, if by chance the packet happens naturally to end with a maximum length 0xFF code block then there is no need to logically append a zero in order to make it encodable. This optimization can be performed without ambiguity, because the receiver will observe that the decoded packet does not end with a zero, and hence will realize that in this case there is no trailing zero to be discarded in order to recreate the original input data. In practice, the gain achieved by this optimization is very slight, but it is useful from a theoretical standpoint. It simplifies analysis of the algorithm's behavior by allowing us to say that COBS encoding adds "no more than one byte of overhead for every 254 bytes of packet data." Without this optimization, that statement would not be true for packets containing an exact multiple of 254 bytes and no zeroes; these packets would incur one extra byte of overhead.

C. Behavior of COBS

COBS encoding has low overhead (on average 0.23% of the packet size, rounded up to a whole number of bytes) and furthermore, for packets of any given length, the amount of overhead is virtually constant, regardless of the packet contents. This section offers an intuitive understanding of why the overhead of COBS is so uniform, in marked contrast to the overhead of PPP and other two-for-one byte stuffing schemes, which are extremely variable.

COBS encoding has low overhead because, in most cases, the size of each code block is exactly the same as the size of the data sequence it encodes. For all of the code blocks 0x01 to 0xFE, the code block contains one code byte and (n–1) non-zero data bytes. The data represented by that code block is the (n–1) data bytes followed by a single zero byte at the end. Hence n input bytes are encoded using exactly n output bytes, so the output size is the same as the input size. Even for code block 0xFF where there is no implied zero and 255 output bytes are used to encode 254 input bytes, the overhead is just 0.4%.

COBS encoding overhead has very low variability compared to other byte stuffing algorithms like PPP. To see why this is so we need to understand how overhead is generated in byte stuffing algorithms. For any byte stuffing algorithm like PPP or COBS, each byte in a packet either generates no overhead, or it generates some fixed amount x. In a given packet of length n bytes, if the proportion of bytes that are overhead-generating is p, then the total overhead for the packet will be npx bytes. For all of today's byte stuffing algorithms, x has some fixed value, but the values of n and p are determined by the size of the packet being encoded and the nature of the data in that packet.

For PPP, overhead is incurred whenever a byte has one of the reserved values 0x7D or 0x7E. On average for uniformly distributed data, only one byte out of every 128 is a reserved value. This occurrence is relatively rare — less than 0.8% of the bytes — but when it does occur the cost is relatively high — an encoding overhead of 100% for that particular byte. All the other byte values — the vast majority — incur no overhead. Thus for PPP x=1 and, for the average packet, $p={}^{1}/_{128}$.

For COBS, the behavior has the opposite characteristics. Overhead is not the rare case; it is the common case. Overhead is not generated by occurrences of the reserved value, it is generated by the large majority of bytes that are *not* the reserved value zero: If more than 254 bytes are encountered without finding a single zero, then one byte of overhead is generated. Fortunately, although overhead is the common case, the one byte of overhead generated is amortized over 254 bytes, making the cost per byte very small. The rare event of encountering a zero can be viewed as an occasional lucky bonus: both the zero and some number of non-zero bytes that preceded it are then encoded with no overhead at all. Thus for COBS the proportion of bytes that are overhead-generating is high — usually the majority of the bytes in the packet — but the amount of overhead each one generates, x, is only ${}^{1}/_{254}$.

Because of these two opposite kinds of behavior, PPP and COBS generate overhead in opposite ways. The encoding overhead of PPP can be characterized as the sum of a series of high-cost low-probability events, whereas the overhead of COBS can be characterized as a series of high-probability low-cost events. For PPP to encode a packet with low overhead, it requires that the proportion of overhead-generating bytes in the packet be small. In the pathological case where every byte in the packet generates overhead, p=1 and the size of the packet doubles. In contrast, COBS already assumes that most bytes generate overhead, so the worst possible case where every byte generates overhead is hardly any different from the average case. For the same reason, the best case for COBS is also hardly any better than the average case. Because x is so small, the final encoded size of the data is very insensitive to the proportion p of bytes that are overhead-generating.

To summarize: The best case for COBS is one byte of overhead, and the worst case is one byte for every 254 bytes of packet data. This is good for large packets, but has an unfortunate side effect for small packets. Every packet 254 bytes or smaller always incurs exactly one byte of overhead, no more, no less.

We regard one byte of overhead a small price to pay in exchange for vastly more predictable overhead and the consequent significant performance gains we get from the ability to send much larger IP packets. However, there could be circumstances where any cost, however small, is unacceptable. To address this concern the next section presents a minor modification to basic COBS called Zero Pair Elimination (ZPE), which exhibits better performance for small packets.

D. Zero Pair Elimination

In experiments on real-world data (see Section V) we observed not only that zero is a common value in Internet traffic, but furthermore that adjacent pairs of zeros are also very common, especially in the headers of small TCP/IP packets. To take advantage of this property of real-world traffic, we created a variant of COBS where the maximum encodable sequence length is slightly shortened, freeing some of the high-numbered codes for other uses. These codes are reassigned to indicate sequences ending with a *pair* of implicit zeroes. Any reduction in the maximum sequence length increases the

Code	Followed by	Meaning
0x00	(not applicable)	(not allowed)
0x01-0xDF	(*n*–1) data bytes	The (*n*–1) data bytes, followed by a single zero
0xE0	223 data bytes	The 223 data bytes, **not** followed by a zero
0xE1-0xFF	(*n*–225) data bytes	The (*n*–225) data bytes, followed by a **pair** of zero bytes

Table 2. Code values used by consistent overhead byte stuffing with zero-pair elimination. Codes 0x01 to 0xDF have the same meaning as in basic COBS, but codes 0xE0 to 0xFF are reassigned to have new meanings.

worst-case overhead, so the maximum length should not be reduced by too much. With our packet traces we found that reassigning 31 codes gives good performance without sacrificing too much of the good worst-case bound that makes COBS useful [Che98]. For other protocols and traffic mixes the 'optimum' cut-off point may be different, but it should be remembered that the benefit of COBS (and COBS/ZPE) encoding is that it is extremely insensitive to the nature of the data being encoded. The penalty for picking the 'wrong' cut-off point is very slight.

In COBS/ZPE codes 0x00 to 0xDF have the same meaning as in basic COBS, and code 0xE0 encodes the new maximum length sequence of 223 bytes without an implicit zero on the end. This change in code assignments gives COBS/ZPE a worst-case overhead of one byte in 223, or about 0.45%. Codes 0xE1 to 0xFF encode sequences that end with an implicit pair of zeroes, containing, respectively, 0 to 30 non-zero data bytes. These new code meanings are summarized in Table 2.

Reassigning some of the codes to indicate sequences ending with a *pair* of implicit zeroes has the good property that now some of the code blocks are *smaller* than the data they encode, which helps mitigate the one-byte overhead that COBS adds. Fig. 4 shows an example of a small packet (actually the beginning of a real IPv4 packet header) that gets one byte smaller as a result of encoding using COBS/ZPE.

The disadvantage of using COBS/ZPE is a slightly poorer worst-case overhead — about 0.45% instead of 0.40% — but this is still a very small worst-case overhead. In fact, as described in Section V, when a typical mix of real-world network traffic is encoded using COBS/ZPE, it actually gets smaller by about 1%. PPP byte stuffing cannot compete with this since PPP never makes any packet smaller.

COBS/ZPE is useful because pairs of zeroes are common in packet headers. Also, the trend towards aligning packet fields on 64-bit boundaries in high-performance protocols sometimes results in padding zeroes between fields. These padding zeroes waste precious bandwidth on slow wireless links. Using COBS/ZPE as the link-layer encoding for these slower links can help to mitigate this effect by encoding these patterns more efficiently. This increased efficiency for zero pairs

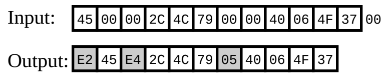
Input:
45 00 00 2C 4C 79 00 00 40 06 4F 37 00
Output:
E2 45 E4 2C 4C 79 05 40 06 4F 37

Fig. 4. Example COBS/ZPE encoding showing input (with a phantom zero logically appended) and corresponding zero-free output. For clarity, COBS code bytes are shown shaded.

makes it more attractive to use a single packet format on all networks, instead of designing different application-layer protocol formats for different speeds of network.

Although for certain packets COBS/ZPE does result in some reduction in size, COBS/ZPE should not be thought of as a general purpose compression algorithm, and it is not intended to compete with more sophisticated (and more computationally expensive) compression algorithms such as Huffman encoding [Huff52] [Knu85] and Lempel Ziv [LZ77] [Wel84]. Although, like PPP, these compression algorithms may have good average performance, for some data they can make the packet bigger instead of smaller [Hum81], and it can be hard to predict how much bigger they may make a packet in the worst case, which is contrary to our goal of ensuring a tight bound on worst-case performance.

It is also not clear that there is a great benefit to applying heavyweight compression at the link layer, since the majority of compressible data is much more effectively compressed before it even reaches the IP layer using data-specific algorithms such as JPEG [ISO10918] for images and MPEG [ISO11172] for video. In the case of data that has been encrypted, it is not possible to apply heavyweight compression at the link layer because data that has been properly encrypted is not compressible [Pra97].

ZPE is a lightweight technique that works well for small packets that contain zero pairs, without sacrificing the primary benefit of COBS: an aggressive bound on worst-case overhead, even for packets that may be large and contain no zero pairs.

E. Lower-Level Framing

COBS is defined in terms of byte operations. This means that there also needs to be some underlying mechanism to detect where the byte boundaries fall in the bit-stream. When used over an RS-232 serial port [RS-232-C], the start/stop bits perform this function (not very reliably [Che98]), at a cost of 20% extra overhead. HDLC framing is usually more efficient, but it too, in the worst case, can add as much as 20% overhead. COBS's twin benefits of guaranteeing to always resynchronize immediately after error and guaranteeing to do so without adding excessive overhead are somewhat diminished if the bit-level framing is not similarly well behaved.

Fortunately, with the choice of an appropriate bit-level framing pattern, COBS-encoded data becomes self-framing, and no separate lower-layer bit stuffing is needed. All that is required to make this work is a bit-level framing pattern chosen so that it can never occur anywhere in COBS-encoded data. If the framing pattern cannot inadvertently appear, there is no need for any bit stuffing mechanism to eliminate it.

Finding an appropriate framing pattern is easy. Since COBS-encoded data contains no zero bytes, we know that there is at least one binary ‘1’ bit somewhere in every byte. This means that in a bit-stream of COBS-encoded data there can be no contiguous run of more than fourteen zero-bits, which suggests a candidate for the COBS end-of-packet marker: a run of fifteen (or more) zero-bits.

The work of Barker [Bar53] and Artom [Art72] gives further guidance about the choice of an appropriate framing pattern. The framing pattern should not only be one that does not appear in the data stream, but also one that cannot appear in any overlap between the marker and the data stream. A framing marker consisting of just a run of zero-bits would not meet this requirement. It could never appear entirely within the encoded packet data, but appending this framing marker to the end of an encoded packet could result in an ambiguous situation. This is because nothing prevents an encoded packet from ending with as many as seven consecutive zero-bits. Upon seeing a long run of zero-bits, the receiver does not have any simple reliable way to tell which, if any, of those zero-bits actually belong to the tail of the previous packet, and which bits belong to the framing marker. This ambiguity can be solved by refining the definition of the framing marker to be a one-bit followed by fifteen zero-bits:

```
1000000000000000
```

Unintended occurrences of the framing marker are now eliminated, because any sequence made by concatenating bits from the end of a packet with bits from the start of the marker will result in a sequence that has at least one one-bit somewhere in the middle. This cannot be mistaken for a framing marker because the framing marker has no one-bits in the middle.

COBS bit-level framing can be defined to use a synchronous idle sequence, or an asynchronous idle sequence, as appropriate to the application requirements. Using a *synchronous* idle sequence, the end-of-packet marker is a one-bit followed by exactly fifteen zero-bits. If the link is idle and there are no packets to transmit, then these 16-bit end-of-packet markers are transmitted back-to-back continuously until there is a packet to transmit. Using an *asynchronous* idle sequence, the end-of-packet marker is a one-bit followed by fifteen *or more* zero-bits. In this case, after the basic end-of-packet marker has been sent, an idle line can simply be filled with as many zero-bits as are needed to bridge the gap until there is another packet to send. So that the receiver knows when the next packet begins, the asynchronous framing marker is defined to be a one-bit, followed by fifteen or more zero-bits, and then another one-bit, as shown below:

```
100000000000000 ... 01
```

For a line that is mostly busy, the synchronous idle specification is marginally better, because the framing marker is only 16 bits long, compared to the 17-bit minimum length of the asynchronous idle framing marker, so there is slightly less overhead.

For a line that is mostly idle, the asynchronous idle specification is marginally better, because packets can be sent with slightly less delay. Using synchronous idle markers, if an idle line has just begun transmitting its next idle end-of-packet marker when an outgoing packet arrives at the interface, the packet will have to wait for 15 bit-times before it can be sent. Using asynchronous idle, the interface could immediately send the start bit and begin sending the packet after a delay of only one bit-time.

Used with either of these framing marker definitions, COBS guarantees a very tight bound, not only on the number of bytes, but also on the total number of bits required to transmit a data packet.

IV. Theoretical Analysis

This section presents a theoretical analysis of the behavior of COBS and COBS/ZPE in comparison to two other techniques, HDLC bit stuffing and PPP byte stuffing. It compares the best-case, worst-case, and average case-encoding overhead, given uniformly distributed random data.

HDLC and PPP are chosen for comparison because both work reliably and are reasonably efficient, and as a result both are widely used in today’s software. They are arguably the canonical examples respectively of bit and byte stuffing.

It is useful to calculate the expected performance for uniform random data, because data that is properly compressed and/or encrypted has a uniform distribution of byte values. Data that is well-compressed must in general have a uniform distribution of byte values, because if it did not then Huffman encoding [Huff52] could be used trivially to compress the data further, contradicting the description of the data as being ‘well compressed’. For similar reasons encryption algorithms seek to maximize the entropy of the data so as to maximize the strength of the encryption [Pra97]. Compressed data already makes up a large part of Internet traffic: For example, World-Wide Web traffic has grown to exceed all other Internet traffic sources combined, whether measured by number of packets, number of bytes, or number of flows [Braun94] [Thom97], and furthermore, 65% of World-Wide Web traffic is compressed GIF or JPEG image data [Gwert96]. We expect to see an increasing awareness of security issues on the Internet in the future, resulting in a similar increase in encrypted data. However, at the present time not all traffic is compressed and/or encrypted, so actual performance on today’s real packets is shown in Section V.

The section concludes with a discussion of the trade-off between minimizing the encoding overhead and minimizing the delay imposed on the stream of data being encoded.

A. Best Case Overhead

The best case for HDLC bit stuffing is a packet that contains no occurrences of five binary ones in a row. In this case, HDLC encoding adds no overhead to the packet at all.

The best case for PPP byte stuffing is a packet that contains no occurrences of the reserved (0x7D or 0x7E) characters. In this case, PPP encoding adds no overhead to the packet at all.

The best case for COBS is a packet with plenty of zeroes, so that nowhere in the packet is there any contiguous sequence of more than 254 non-zero bytes. In this case, each block is encoded with no overhead at all. Counting the phantom zero that has to be added to every packet before encoding, this results in a best-case overhead of a single byte, for any size of packet.

The best case for COBS/ZPE is a packet composed entirely of zeroes. In this case, each pair of zeroes is encoded as a one-byte code, 0xE1, resulting in an approximate halving of packet size. As with basic COBS, a phantom zero is still appended to the tail of every packet, so the best-case encoded size is half of the length *n* of the packet plus one byte, rounded up to an integer number of bytes, meaning packet size is *reduced* by:

$$n - \left\lceil \frac{n+1}{2} \right\rceil = \left\lfloor \frac{n-1}{2} \right\rfloor \text{ bytes}$$

B. Worst-Case Overhead

The worst case for HDLC bit stuffing is a packet that contains nothing but binary ones. In this case, HDLC encoding adds one bit for every five, giving an overhead of 20%.

The worst case for PPP byte stuffing is a packet that consists entirely of reserved (0x7D or 0x7E) characters. In this case, encoding doubles the size of the packet, giving an overhead of 100%.

The worst case for both COBS and COBS/ZPE is a packet that contains no zeroes at all. In the case of basic COBS, each sequence of 254 bytes of packet data is encoded using 255 bytes of output data, giving one byte of overhead for every 254 bytes of packet data. The maximum overhead is $^1/_{254}$ (roughly 0.4%) of the packet size, rounded up to a whole number of bytes. For example, a maximum size IP packet over Ethernet is 1500 bytes long, and in the worst possible case COBS would add an overhead of six bytes to a packet of this size. In the case of COBS/ZPE, each sequence of 223 bytes of packet data is encoded using 224 bytes of output data, giving one byte of overhead for every 223 bytes of packet data. The maximum overhead is $^1/_{223}$ (roughly 0.45%) of the packet size, rounded up to a whole number of bytes. For example, in the worst possible case COBS/ZPE would add an overhead of seven bytes to a maximum size Ethernet packet.

C. Expected Overhead

Knowing each algorithm's extremes is useful, but it is also useful to know the average overhead because that tells us how efficiently the algorithm uses the underlying medium, in the sense of what overall proportion of its intrinsic capacity we expect to be available for carrying data and what proportion we expect to be consumed by encoding overhead. Since byte stuffing is a process that takes as input a packet composed of characters from an alphabet of 256 possible symbols and gives as output a packet composed of characters from an alphabet of only 255 possible symbols, in general there must be some overhead. Exactly how much longer a particular packet becomes may or may not depend on the contents of that packet, depending on the algorithm being used. With some algorithms there may be fortunate packets that incur no overhead at all, but information theory tells us that for random data the average overhead must be at least:

$$\frac{\log 256}{\log 255} - 1 \approx 0.0007063, \text{or roughly } 0.07\%$$

This theoretical bound gives us a metric against which to judge different byte stuffing schemes. Some algorithms may be able to beat this bound for some packets, but there is no algorithm whose average performance over all packets can beat this bound. Perhaps more significantly, this average performance bound tells us that no algorithm can ever have a worst-case bound better than 0.07063%. If any algorithm did have a worst-case bound better than this, that would necessarily mean that its average performance over all inputs would also be better than 0.07063%, and we know that cannot be the case.

In practice we can take advantage of the fact that our network traffic is not purely random data, by recognizing that certain patterns of data occur more frequently than others and devising an algorithm that encodes this particular mix of data more efficiently. This allows us to improve our average performance for typical network traffic, but it does not change the fact that the worst case can never be improved beyond the theoretical limit.

C.i. Expected Overhead for HDLC

In [Pap87] Papastavridis shows that in a prefix-synchronized code such as HDLC, if the data is *n* bits long, the prefix is *k*+1 bits long, and *n*»*k*, the mean number of stuffing bits is approximately:

$$\frac{n}{2^k - 1}$$

For HDLC, the prefix referred to is the sequence 0111111, so *k*=6. Thus Papastavridis shows that the mean number of stuffing bits in HDLC-encoded data is roughly one bit in 63. Note that Papastavridis shows his result in terms of the proportion of stuffing bits expected to be *observed in the final output data*, not in terms of the number of stuffing bits the encoding algorithm is expected to *add to the input data*. If in the output one bit in 63 is a stuffing bit, that is equivalent to saying that the encoding algorithm added one bit for every 62 bits of input data. Hence, in terms consistent with those used throughout the rest of this paper, HDLC bit stuffing adds an average overhead of one bit in 62, or a little over 1.6%.

C.ii. Expected Overhead for PPP

For PPP the expected overhead is easy to calculate. PPP has only two distinct behavior patterns: it either reads a single byte and writes a single byte, or it reads a single byte and writes a pair of bytes. In uniformly distributed data, the probability that any given byte will be one of PPP's two reserved values, causing PPP to output two bytes instead of one, is $^2/_{256}$. In a packet of length *n*, there will be on average $n \times {}^2/_{256}$ occurrences of reserved values and $n \times {}^{254}/_{256}$ occurrences of other values, giving an expected output length of:

$$n \times 2 \times \tfrac{2}{256} \; + \; n \times 1 \times \tfrac{254}{256} \; = \; 1.0078125n$$

An expected output length 1.0078125 times the input length gives an expected overhead of 0.78125%, about 11 times worse than the theoretical optimum.

C.iii.Expected Overhead for COBS

For COBS the average overhead is a little harder to calculate than for PPP, since COBS has 255 different behaviors, rather than just the two that PPP has. In addition, each behavior not only writes a different number of output bytes, each behavior also reads a different number of input bytes. Since the number of input bytes read is not always one as it is for PPP, we must also calculate the average number of bytes read per code block, and divide the average output by the average input to determine the overall average overhead.

First we determine the average input per code block by multiplying the number of bytes each code block consumes by the probability of that code block occurring, and summing the results. Likewise we determine the average output per code block by doing the same calculation using the number of bytes each code block generates. The ratio of average output divided by average input gives us the average encoding overhead.

If the first byte the algorithm encounters is a zero, then that single byte is encoded as a code block. The input in this case is one byte and the output is one byte and the probability p(0x01) of this happening is $^{1}/_{256}$. If the first byte is not a zero, but the second byte is, then the algorithm reads two bytes and outputs two bytes and the probability p(0x02) of this happening is $^{255}/_{256} \times {}^{1}/_{256}$. The probability p($n$) that the encoder reads n–1 non-zero bytes ($n \leq 254$) followed by a zero is:

$$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256}$$

The longest code block, code 0xFF, occurs when the algorithm encounters 254 non-zero bytes without seeing a single zero. The probability p(255) of this happening is $({}^{255}/_{256})^{254}$, and in this case the algorithm reads 254 bytes and writes out a block of 255 bytes.

The ratio of average output divided by average input is:

$$\frac{\left(\sum_{n=1}^{254} n \times \mathrm{p}(n)\right) + 255 \times \mathrm{p}(255)}{\left(\sum_{n=1}^{254} n \times \mathrm{p}(n)\right) + 254 \times \mathrm{p}(255)} \approx 1.002295$$

The theoretical average overhead for COBS on random data is therefore a little under 0.23%. This is about $3^{1}/_{4}$ times worse than the theoretical optimum, more than three times better than PPP’s average.

C.iv.Expected Overhead for COBS/ZPE

As with basic COBS, we calculate the probability of each code block, and the number of bytes of input and output for each code block. The probability p(0x01) that the first byte is a zero and the second is not, resulting in a code 0x01 block, is $^{1}/_{256} \times {}^{255}/_{256}$. The probability p(0xE1) that both the first and the second bytes are zero, resulting in a code 0xE1 block, is $^{1}/_{256} \times {}^{1}/_{256}$. Table 3 enumerates, for each code of the form k+n, the probability of code block k+n occurring, and the number of input and output bytes for that code block.

Base Code k	Offset n	Probability p(k+n)	Input in(k+n)	Output out(k+n)
k = 0x00	1≤n<0x20	$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256} \times \frac{255}{256}$	n	n
k = 0x00	0x20≤n<0xE0	$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256}$	n	n
k = 0x00	n = 0xE0	$\left(\frac{255}{256}\right)^{n-1}$	$n - 1$	n
k = 0xE0	1≤n<0x20	$\left(\frac{255}{256}\right)^{n-1} \times \frac{1}{256} \times \frac{1}{256}$	$n + 1$	n

Table 3. Code block probabilities for COBS/ZPE.

The ratio of average output divided by average input is:

$$\frac{\sum_{n=1}^{255} \mathrm{p}(n) \times \mathrm{out}(n)}{\sum_{n=1}^{255} \mathrm{p}(n) \times \mathrm{in}(n)} \approx 1.002800$$

The theoretical average overhead for COBS/ZPE on random data is therefore about 0.28%. This is a little higher than basic COBS, which is not surprising: The maximum length code block is about 12% shorter, making it likely to occur more often, and every time it does occur the amount of overhead if adds is about 12% higher (0.45% instead of 0.40%).

D. Encoding Delay

One possible criticism of COBS encoding is that it appears to add delay to data transmission, whereas HDLC and PPP do not. The argument is that as each bit is fed to an HDLC encoder, the encoder immediately generates its corresponding output, namely that bit, or that bit followed by a zero stuffing bit, as appropriate. Likewise, as each byte is fed to a PPP encoder, the encoder immediately generates its corresponding output, namely that byte, or a pair of bytes, as appropriate. In both cases, for each unit of data the output generation is immediate. The encoder does not have to wait until it has seen some amount of subsequent input before it is able to decide what its output will be. In contrast, a COBS encoder may have to buffer up to 254 bytes of data internally before it generates any output, which suggests that COBS encoding could add a significant amount of delay to the data transmission pipeline.

This section addresses this possible criticism in two ways. The first is to show that delay and worst-case overhead are intrinsically linked, so any delay in the COBS algorithm is not so much a flaw peculiar to the COBS algorithm, but a necessary consequence of achieving a low bound on worst-case overhead. The second is that the apparent dependency on future data, and the consequent apparent encoding delay, are due to considering bytes as the basic unit of data. If we consider the network packet as the basic unit of data, then there is no dependency on future data, and thus no encoding delay.

D.i. Relationship Between Delay and Worst-Case Overhead

Any data stuffing algorithm is effectively a finite state machine that reads input, undergoes state changes, and writes output. Since in the worst case the output is necessarily longer than the input, for each *n* units of data read some number *m*>*n* units of data may have to be written. To minimize worst-case overhead, we should make *m* as small as possible. Unless we plan to allow reading and writing of fractional units of data, which is beyond the scope of this paper, *n* and *m* must be integers, and the smallest integer greater than *n* is *n*+1. Thus the worst-case encoding ratio can be no better than $^{n+1}/_{n}$. Consequently, to minimize worst-case overhead, we should make *n* as large as possible. The smaller the value of *n*, the larger the worst-case encoding overhead. For PPP the amount of data read before an output is generated is always one byte, so *n*=1, and the worst-case encoding ratio is 2:1. For COBS the amount of data read before an output is generated may be up to 254 bytes, so *n*=254, and the worst-case encoding ratio is 255:254.

Upon casual inspection it may appear that HDLC is able to beat this performance limit, since its worst-case overhead is only 20%, yet it is always able to generate an output for every single bit of input. However, this interpretation is misleading, because HDLC encoding transfers the delay to the receiving end of the pipeline. While the HDLC encoder is able to process individual bits with no delay, the HDLC decoder is not. When the HDLC decoder observes a series of one-bits, it is not able to say whether those one-bits are data bits or the end-of-frame marker until it has examined the sixth bit to see whether it is a zero or a one. Hence in the case where it receives five ones followed by a zero, the HDLC decoder has to read six bits before it can generate any output, at which time it can then immediately generate five bits of output. Hence HDLC does adhere to the general delay/overhead rule described above: it has a worst-case encoding ratio of 6:5, and consequently there is a necessary delay of up to five bits before the true meaning of a particular bit can be properly determined by the receiver.

To have low overhead we need to have a large value of *n*, and to have a large value of *n* we need to add delay, at the transmitter and/or at the receiver. This suggests the unhappy conclusion that it is impossible to have a small worst-case overhead without also adding a large amount of delay but, as shown below, this apparent problem may not matter at all.

D.ii. Units of Data for Encoding Algorithms

At the beginning of this section, we stated that both HDLC and PPP encoders generate immediate output for each unit of input they are given. However, for HDLC the unit of input was a single bit, while for PPP the unit of input was the byte. If we consider the input to the PPP encoder to be a stream of bits instead of bytes, then its output is no longer always immediate. Upon being presented with the first bit, the PPP encoder may be unable to generate any output. It may have to wait until it has received the next seven bits to complete the entire byte before it knows what output it should generate. However, in practice, PPP encoders are not presented with data one bit at a time. They are presented with data at least a byte at a time. In fact, in most networking software, the PPP encoder is not even presented with single bytes; it is presented with an entire packet to encode, and the entire packet is encoded in a single program loop. One of the reasons for processing data a packet at a time is that popular network protocols such as Ethernet [Tan88] [IEEE802.3] and IP [RFC791] do not carry data as an unstructured stream of bits or bytes over virtual (or physical) circuits, but as packets, and packets are the atomic units of data delivery.

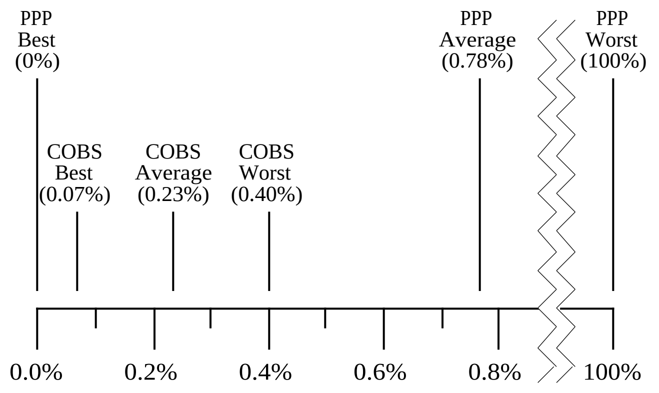

Fig. 5. Encoding overhead for a 1500 byte packet. PPP's best, average, and worst cases vary widely, in contrast to COBS's best, average and worst cases, which fall within a much narrower range.

In the same way that PPP encoding is more usefully applied at a per-byte level than at a per-bit level, COBS encoding is more usefully applied at a per-packet level than at a per-byte level. Although the COBS encoding of a given byte in general depends on the values of other nearby bytes, the encoding of any given packet is idempotent, and is not dependent on knowledge of any previous or subsequent packets. Since the kinds of networks considered in this paper use packets as their basic units, it is reasonable to relax the restriction that the byte stuffing process should be an idempotent per-byte operation. It is perhaps unsurprising that giving a byte stuffing algorithm access to more data allows it to perform better.

E. Summary

Fig. 5 shows a comparison of the results for PPP and for COBS, and visually illustrates the dramatic difference between the narrow range of overheads generated by COBS and the wide range of overheads that can be generated by PPP.

These average results hold for well-compressed packets which contain a uniform distribution of byte values, but not all Internet traffic is well-compressed. In addition, it is not possible to have fractional bytes of overhead. In theory a 40-byte IPv4 TCP acknowledgement packet encoded with COBS may average an overhead of $40 \times 0.23\% = 0.092$ bytes, but in practice that fraction is rounded up to an entire byte of overhead. For small packets this rounding up may be a more dominant contributor to overhead than the actual underlying properties of the algorithm. To investigate how much effect this potential problem might have, we encoded traces of real-world network traffic using both COBS and PPP byte stuffing, and these results are presented in the next section.

V. Experimental Results

This section presents the encoding behavior of COBS and other protocols for real-world network traffic. Real-world network traffic may not behave the same way as Section IV's theoretical traffic. In real-world traffic, small packets are common, and not all data is compressed and/or encrypted. To see how these factors affect the algorithms, we gathered traces of network traffic using tcpdump [Jac89] and compared how efficiently those packets were encoded by HDLC, PPP, COBS and COBS/ZPE.

We captured traces of traffic sent both over Metricom packet radio interfaces [Che96] and over Ethernet. The reason for studying packet radio traffic as well as Ethernet traffic was because unlicensed FCC Part 15 radio devices were one of the main motivations behind the development of consistent overhead byte stuffing. The performance of wide-area radio interfaces is dramatically worse than that of an Ethernet (both in bandwidth and in latency) and this performance difference may have an influence on the way people use their network connection. Consequently, we wanted also to study traffic patterns in that environment as well as in the more common Ethernet environment.

Two traces are presented here. The first is a wireless traffic trace, consisting of predominantly small packets. Just over half of the packets (51%) in this trace are either TCP ack packets containing no data, or one-byte TCP data packets, such as a single keystroke on a remote login connection. The second trace is a large file transfer containing roughly two thirds maximum-sized TCP data packets and one third TCP acks. (63.6% of the IP packets were maximum-sized TCP data packets and 36.3% were TCP acks.)

For each trace, the packets were encoded using HDLC, PPP, COBS and COBS/ZPE. For each method a histogram shows overheads in the range ±30 bytes on the horizontal axis, and for each of those overheads, the percentage of packets incurring that overhead. This percentage is plotted on a log scale to show more detail at the low end. For these traces 0.001% of the packets amounts to less than one packet, and since the only number of packets less than one is zero packets, 0.001% is chosen as the base line of the log scale.

In all of our traces, PPP's worst-case encoding was always significantly higher than COBS's worst case. Even so, PPP never came close to using the full factor-of-two overhead that is possible. Nevertheless, the fact that we never observed these pathological packets does not mean that it is safe to engineer a PPP implementation that does not take them into account. There is at least one example of real-world traffic that exhibits frequent reproducible pathological PPP encoding behavior: voice over IP using G.721 [G721]. During periods of near-silence, this encoding can sometimes transmit data that is almost entirely 0x7E byte values, causing pathological PPP data expansion [Carl97].

A. Three-Day Trace

One of our colleagues frequently works at home, and his sole Internet connection is via a portable ISM-band packet radio attached to his laptop computer. We collected a trace of all his packets for a period of three days. The goal was to capture a representative trace of packets from a user who makes extensive use of a wireless interface. The trace contains 36,744 IP packets, totalling 10,060,268 bytes of data (including IP headers and higher layers; not including the link-level header). The MTU of the wireless interface in this case was 1024 bytes, giving a worst-case COBS overhead for large packets of five bytes.

However, most of the packets captured were not large; 69% of the packets were shorter than 254 bytes and necessarily incurred exactly one byte of overhead when encoded with COBS. Moreover, 41% of the packets were exactly 40 bytes long, which is just the length of a TCP acknowledgement containing no data. Another 10% of the packets were exactly 41 bytes long, which is the length of a TCP packet containing just one data byte. Taking these two numbers together, this means that over half the packets were 40 or 41 bytes long. Only 15% of the packets were maximum-sized 1024-byte packets.

The three-day trace is a particularly challenging test case with which to evaluate COBS, because it contains so many small packets. The results for this trace file are shown in Fig. 6.

HDLC incurred a total overhead of 703,607 bits (0.87%). Over 75% of the packets incurred from one to ten bits of overhead, but a few packets incurred over 300 bits of overhead.

PPP incurred a total overhead of 36,069 bytes (0.36%). 74% of the packets incurred no overhead, but some packets incurred a significant amount. More than 100 packets incurred 15 bytes of overhead or more, and one packet fell beyond the scale of the graph with an overhead of 53 bytes. In this trace no packets incurred more than 53 bytes of overhead, supporting the belief that although conventional byte stuffing forces us to design for a factor-of-two safety margin, in practice that safety margin is almost entirely unused.

For COBS the overhead is concentrated in a tight spike in the middle: every packet incurred one to four bytes of overhead. COBS incurred a total overhead of 57,005 bytes (0.57%), meaning that even in this unfavorable test case COBS costs only 0.21% extra compared to PPP, for the benefit of having a tight bound on the worst-case overhead. 74% of the packets had exactly one byte of overhead, 7% had two bytes, 8% had three bytes, and 11% had four.

COBS/ZPE maintained a tight bound on worst-case performance while doing on average much better than either PPP or COBS. For a 1024-byte packet the maximum possible COBS/ZPE overhead is five bytes, but in fact in this trace no packet incurred more than four. In addition COBS/ZPE *reduced* the overall size of the data by 26,238 bytes, giving a net overall saving of 0.26%.

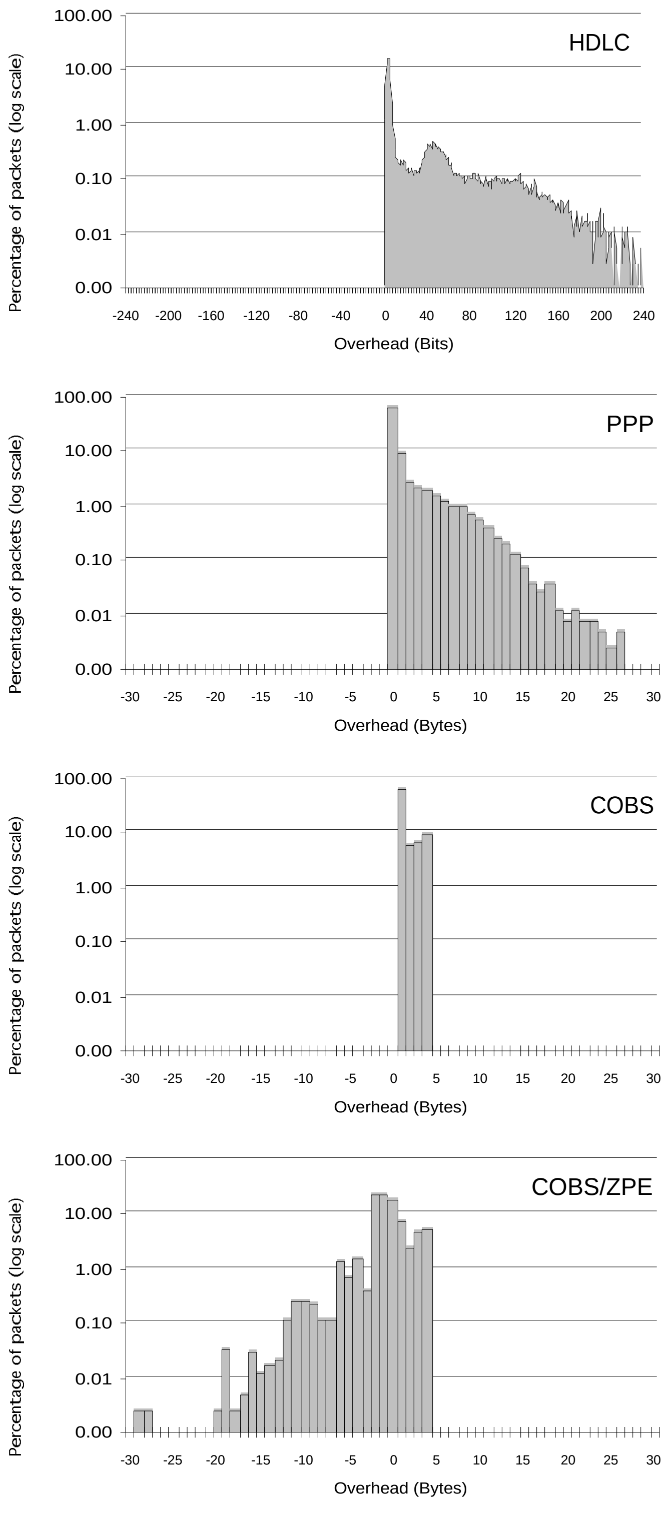

Fig. 6. Encoding overhead distribution for three-day trace. Histograms showing, for each amount of overhead indicated on the horizontal axis, the percentage of packets that incur that overhead. All histograms are drawn to the same scale and labelled in bits or bytes as appropriate.

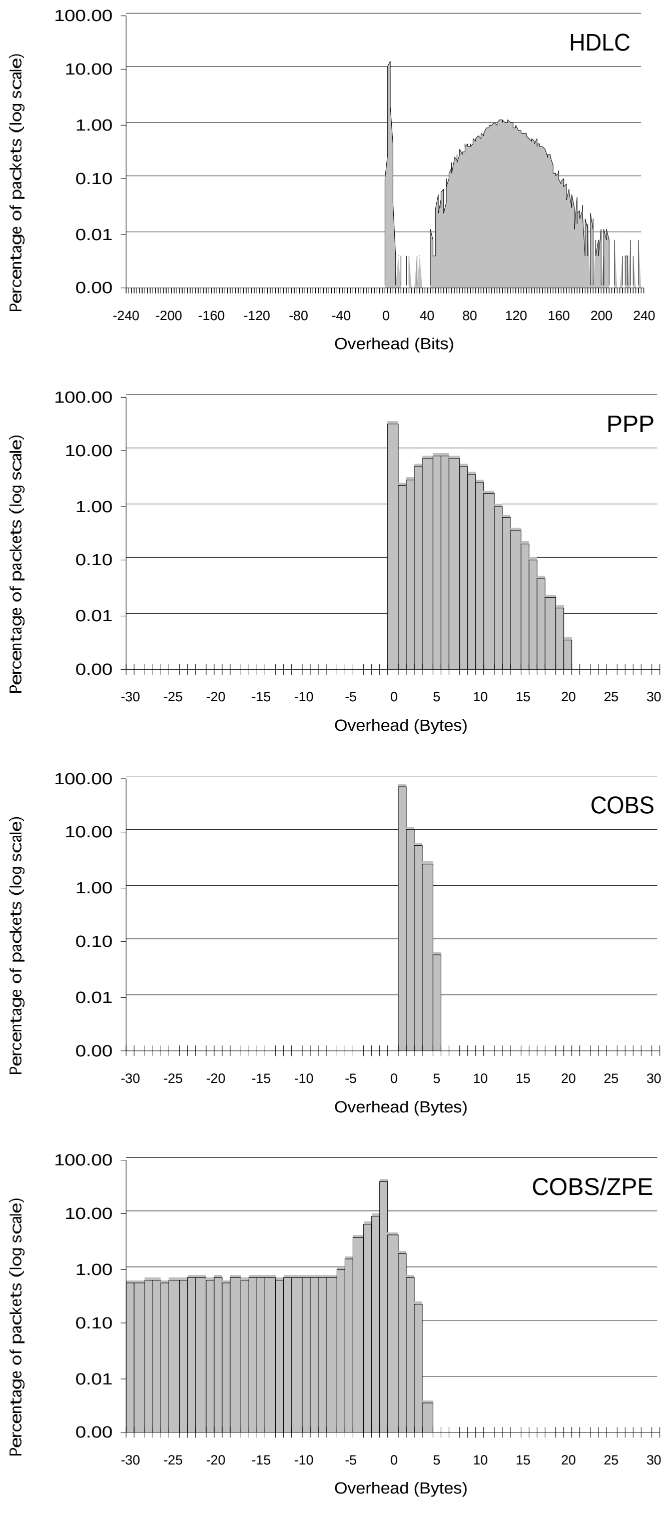

Fig. 7. Encoding overhead distribution for MPEG trace. Histograms showing, for each amount of overhead indicated on the horizontal axis, the percentage of packets that incur that overhead. All histograms are drawn to the same scale and labelled in bits or bytes as appropriate.

B. MPEG Trace

With the increasing popularity of the World Wide Web, we might expect to see large packets and compressed data (particularly image data) becoming more common on the Internet. To see how COBS would perform under these conditions we captured a large bulk transfer of compressed image data. The data file was MIRV.MPG, a 15.3MB MPEG [ISO11172] file of an MTV music video, and it was transferred using ftp [RFC959]. The trace contains 25,858 IP packets, totalling 18,269,430 bytes of data. The MTU of the wireless interface was 1088 bytes, giving a worst-case COBS overhead for large packets of five bytes.

The MPEG trace is more favorable to COBS because it contains many large packets. 63% of the packets were maximum-sized IP packets, 1088 bytes long. The results for this trace file are shown in Fig. 7.

HDLC incurred a total overhead of 1,862,796 bits (1.27%) and this histogram has two visibly separate peaks. The first peak, one to eight bits, is the result of the overhead from short (40-byte) ack packets. The second peak, 40-200 bits, is the result of the overhead from the long (1088-byte) data packets.

PPP incurred a total overhead of 101,024 bytes (0.55%). 36% of the packets (mostly the ack packets) incurred no overhead. The majority of packets incurred one to ten bytes of overhead and one packet incurred as much as 20 bytes of overhead. In this trace no packet incurred more than 20 bytes of overhead, supporting the belief that conventional byte stuffing's required factor-of-two safety margin remains almost entirely unused in practice.

COBS incurred a total overhead of only 35,410 bytes (0.19%), and the overhead is concentrated in a tight spike in the middle: every packet incurred between one and five bytes of overhead. 77% of the packets had exactly one byte of overhead. Only 17 packets in the entire trace incurred five bytes of overhead, and as expected, no packets incurred more than that.

COBS/ZPE maintained a tight bound on worst-case performance while doing on average much better than either PPP or COBS. For a 1088-byte packet the maximum possible COBS/ZPE overhead is five bytes, but in fact in this trace no packet incurred more than four bytes of overhead. In addition COBS/ZPE *reduced* the overall size of the data by 161,548 bytes, giving a net overall saving of 0.88%.

VI. Conclusions

COBS is a useful addition to our arsenal of techniques for data communications. It is computationally cheap, easy to implement in software, and gives significant performance benefits by allowing much larger packets to be sent over a given piece of network hardware.

COBS is easy to implement efficiently in software, even for primitive microprocessors. In one project at Stanford COBS has been implemented in hand-written eight-bit assembly code to allow a small embedded control device to connect to a wireless interface and communicate with Internet hosts using UDP/IP [Pog96]. The device needs to be able to send and receive one-kilobyte blocks of data but does not have enough memory for either an implementation of TCP or of IP fragmentation and reassembly. Without COBS it would have been much harder to make the device work. We would have had to add extra memory to the device and would have had to do a lot of extra development work to implement TCP and/or IP fragmentation and reassembly in eight-bit assembly code.

In retrospect it is surprising that COBS or similar techniques have never been described in the literature before. Perhaps one reason is that, until the development of unlicensed radio transmitters under the FCC's Part 15 ISM band rules, the networking community had not confronted the problem of dealing with devices where the maximum transmission size is a hard limit dictated at the physical level.

COBS has the potential to set a new standard, not only for packet radio applications, but for all future applications that require framing and/or byte stuffing.

The benefit of conventional two-for-one substitution encodings like PPP, compared to COBS, is that they may encode small packets with no overhead whereas basic COBS always adds exactly one byte. However, three factors make this apparent benefit of conventional byte stuffing algorithms less compelling.

The main factor is that the pressing problem for many new wireless devices is that of sending large packets, not small packets. It is the large packets that cause problems, because the software must be able to ensure that they do not exceed the device's physical and regulatory limits.

Another factor is that the move to IPv6 [RFC1883] in the future means that the very smallest packets, where PPP does better than COBS, will become increasingly uncommon. In addition, header compression techniques exist to reduce the overhead of the packet headers (especially over slow links) [Deg96], and those header compression techniques reduce the header size by amounts measured in tens of bytes, dwarfing concerns about differences of a single byte here and there.

Finally, if even a single byte of overhead is unacceptable, a trivial modification to COBS to support Zero Pair Elimination makes it perform better than PPP, even for short packets. COBS/ZPE beats both PPP's average overhead and its worst-case overhead.

Although COBS is described in this paper in terms of eight-bit bytes, the COBS principle can also be applied to other word lengths. Further information about this and other aspects of COBS is available in [Che98].

Acknowledgement

The authors express their sincere thanks to Craig Partridge for his many substantial suggestions and contributions to this work, to the anonymous SIGCOMM '97 reviewers for their constructive feedback on the SIGCOMM paper from which this paper is derived, and to the anonymous TRANSACTIONS reviewers who helped improve it still further.

APPENDIX

Source Code Listings

```
/*
 * StuffData byte stuffs “length” bytes of
 * data at the location pointed to by “ptr”,
 * writing the output to the location pointed
 * to by “dst”.
 */

#define FinishBlock(X) (*code_ptr = (X),   \
                        code_ptr = dst++, \
                        code = 0x01       )

void StuffData(const unsigned char *ptr,
unsigned long length, unsigned char *dst)
    {
    const unsigned char *end = ptr + length;
    unsigned char *code_ptr = dst++;
    unsigned char code = 0x01;

    while (ptr < end)
        {
        if (*ptr == 0) FinishBlock(code);
        else
            {
            *dst++ = *ptr;
            code++;
            if (code == 0xFF) FinishBlock(code);
            }
        ptr++;
        }
    FinishBlock(code);
    }
```

Listing 1. COBS encoding in C.

```
/*
 * UnStuffData decodes “length” bytes of
 * data at the location pointed to by “ptr”,
 * writing the output to the location pointed
 * to by “dst”.
 */

void UnStuffData(const unsigned char *ptr,
unsigned long length, unsigned char *dst)
    {
    const unsigned char *end = ptr + length;
    while (ptr < end)
        {
        int i, code = *ptr++;
        for (i=1; i<code; i++) *dst++ = *ptr++;
        if (code < 0xFF) *dst++ = 0;
        }
    }
```

Listing 2. COBS decoding in C.

REFERENCES

[ARRL84] *AX.25 Amateur Packet-Radio Link-Layer Protocol Version 2.0*, October 1984. Available from the American Radio Relay League, Newington CT USA 06111.

[Art72] A. Artom. Choice of Prefix in Self-Synchronizing Codes, *IEEE Transactions on Communications*, vol. COM-20, pp. 253-254, April 1972.

[Bar53] R. H. Barker. Group Synchronizing of Binary Digital Systems, *Communication Theory (Proceedings of the Symposium on the Applications of Communication Theory, September 1952).* W. Jackson, Ed. New York: Academic Press, 1953.

[Ben82] Chris Bennett. The Overheads of Transnetwork Fragmentation. *Computer Networks*, vol. 6, pp. 21-36, February 1982.

[Braun94] H. W. Braun and K. Claffy. Web Traffic Characterization: an assessment of the impact of caching documents from NCSA’s web server. *Proceedings of Second International World Wide Web (WWW) Conference ’94*. Chicago, IL, October 1994.

[CA-96.26] CERT^{SM} Advisory CA-96.26. *Denial-of-Service Attack via ping*, December 1996.

[Carl97] Personal communication, James Carlson, Principal Software Engineer, IronBridge Networks, September 1997.

[Che96] Stuart Cheshire and Mary Baker. Experiences with a Wireless Network in MosquitoNet. *IEEE Micro*, February 1996. An earlier version of this paper appeared in *Proceedings of the IEEE Hot Interconnects Symposium ’95*, August 1995.

[Che97] Stuart Cheshire and Mary Baker. Consistent Overhead Byte Stuffing. *Proceedings of ACM SIGCOMM 1997*, September 1997.

[Che98] Stuart Cheshire. *Consistent Overhead Byte Stuffing.* Ph.D. dissertation, Stanford University, Stanford, California, March 1998.

[Deg96] Mikael Degermark, Mathias Engan, Björn Nordgren, Stephen Pink. Low-Loss TCP/IP Header Compression for Wireless Networks. *Proceedings of MobiCom ’96,* Rye, New York, November 10-12, 1996.

[ECMA-40] European Computer Manufacturers Association Standard ECMA-40: HDLC Frame Structure, December 1973, Sept. 1976 & Dec. 1979.

[G721] CCITT Recommendation G.700-G.795 (1988), *General Aspects of Digital Transmission Systems, Terminal Equipment*. (‘Blue Book’)

[Gwert96] James Gwertzman and Margo Seltzer. World-Wide Web Cache Consistency. *Proceedings of USENIX, 1996.* January 1996.

[Huff52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Communication Theory (Proceedings of the Symposium on the Applications of Communication Theory, September 1952).* W. Jackson, Ed. New York: Academic Press, 1953.

[Hum81] Pierre Humblet. Generalization of Huffman Coding to Minimize the Probability of Buffer Overflow. *IEEE Transactions on Information Theory*, vol. IT-27, pp. 230-232, March 1981.

[IEEE802.3] *Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*. Institute of Electrical and Electronic Engineers, IEEE Standard 802.3-1990, 1990.

[ISO10918] ISO Committee Draft 10918. *Digital compression and coding of continuous-tone still images*, ISO/IEC 10918, 1991.

[ISO11172] ISO Committee Draft 11172. *Information Technology — Coding of moving pictures and associated audio for digital storage media up to about 1.5 Mbit/s*, ISO/IEC 11172-1, 1993.

[Jac89] V. Jacobson, C. Leres, and S. McCanne. *tcpdump*, <ftp://ftp.ee.lbl.gov/tcpdump. tar.Z>, June 1989.

[Kent87] C. Kent and J. Mogul. Fragmentation Considered Harmful. *Proc. of SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology*. August, 1987.

[Knu85] D. E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms*, vol. 6, pp. 163-180, 1985.

[LZ77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, May 1977.

[Pap87] Stavros Papastavridis. On the Mean Number of Stuffing Bits in a Prefix-Synchronized Code, *IEEE Transactions on Communications*, vol. COM-35, pp. 453-455, April 1987.

[Pog96] Elliot Poger. *Radioscope*, < http://www.poger. com/RadioScope/>, Hardware project for Stanford University Class EE281 "Embedded System Design Laboratory", December 1996.

[Pra97] Personal communication, Vaughan Pratt, August 1997.

[RFC791] J. Postel. *Internet Protocol.* STD 5, RFC 791, September 1981.

[RFC959] J. Postel, J. Reynolds. *File Transfer Protocol (FTP).* RFC 959, October 1985.

[RFC1055] J. Romkey. *A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP*. RFC 1055, June 1988.

[RFC1122] R. Braden. *Requirements for Internet Hosts — Communication Layers.* RFC 1122, Oct. 1989.

[RFC1135] J. Reynolds. *The Helminthiasis (infestation with parasitic worms) of the Internet.* RFC 1135, December 1989.

[RFC1191] J. Mogul and S. Deering. *Path MTU Discovery.* RFC 1191, November 1990.

[RFC1662] William Simpson. *PPP in HDLC-like Framing.* RFC 1662, July 1994.

[RFC1883] Steve Deering and Bob Hinden. *Internet Protocol, Version 6 (IPv6) Specification.* RFC 1883, December 1995.

[RS-232-C] Electronic Industries Association Standard RS-232-C: *Interface between Data Terminal Equipment and Data Communication Equipment Employing Serial Binary Data Interchange,* October 1969.

[Tan88] Andrew S. Tanenbaum, *Computer Networks, 2nd ed.* Englewood Cliffs, N.J.: Prentice-Hall, c1988.

[Thom97] Kevin Thompson, Gregory J. Miller, and Rick Wilde. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network.* November/December 1997.

[US94-15] United States Title 47 Code of Federal Regulations (CFR), Federal Communications Commission (FCC) Part 15, Low-Power Device Regulations. U.S. Government Printing Office.

[Wel84] T. Welch. A Technique for High-Performance Data Compression. *Computer*, June 1984.

Stuart Cheshire received the B.A. and M.A. degrees from Sidney Sussex College, Cambridge, U.K., in June 1989 and June 1992, and the M.Sc. and Ph.D. degrees from Stanford University, Stanford, CA, in June 1996 and April 1998.

He previously worked on IBM Token Ring with Madge Networks, Little Chalfont, U.K., and is currently a Senior Scientist with Apple Computer, Cupertino, CA, specializing in Internet Protocols. He has previously published papers in the areas of wireless and networking, and Mobile IP.

Dr. Cheshire is a member of the Association of Computing Machinery (ACM).

Mary Baker (S'93–M'94) received the B.A., M.S., and Ph.D. degrees from the University of California in Berkeley.

She is currently an Assistant Professor in the Computer Science Department, Stanford University, CA, where she is leading the development of the MosquitoNet Mobile and Wireless Computing Project and the Mobile People Architecture. Her research interests include operating systems, distributed systems, and software fault tolerance.

Dr. Baker is the recipient of an Alfred P. Sloan Research Fellowship, a Terman Fellowship, a National Science Foundation (NSF) Faculty Career Development Award, and an Okawa Foundation Grant. She is a member of the Association of Computing Machinery (ACM).