Operating System Support for Protocol Boosters

A. Mallet, J. D. Chung and J. M. Smith[1]

{ale,jdchung,jms}@cis.upenn.edu

Distributed Systems Laboratory

University of Pennsylvania, Philadelphia, PA 19104-6389

Abstract

"Protocol Boosters" are modules inserted into protocol graphs. They allow the protocol's behavior to adapt to its environment. Boosters can mask undesirable properties of links or subnets in an internetwork. The method permits use of proprietary protocols and supports end-to-end optimizations.

We have implemented Protocol Boosters support in the FreeBSD version of UNIX for Intel architecture machines. Our prototype embeds boosters in the 4.4 BSD-Lite Internet Protocol (IP) stack. We have measured the performance of two prototype boosters: an encryption booster (for passage across insecure subnets) and a compression booster (for passage across bandwidth-impaired subnets).

Our measurement data suggests that OS support for this method can be constructed with low performance overhead; execution of the protocol elements dominates any overhead introduced by our implementation. We discuss some lessons learned from the implementation.

1 Introduction

Network protocols are designed to meet application requirements for data communications, including security, reliability and performance. The dominant design and implementation process for protocols has been to first enumerate the requirements for the protocol, and then design a protocol that provides the necessary features end-to-end[16]. The protocol is then optimized by identifying common cases and implementing fast paths for these cases; TCP/IP is an example[4]. The resulting protocol is robust end-to-end and typically provides good performance. Extremely poor performance can result when the assumptions permitting fast path execution are not met.

1.1 Protocol Boosters

Protocol graphs[11] are a means of representing the interactions between protocol elements which carry out functions required by the protocol, *e.g.*, round-trip time estimation. An approach initially suggested by Feldmeier, *et al.*[8], is the design of "Protocol Boosters." Protocol Boosters are protocol elements intended to be transparently inserted into and deleted from protocol graphs on an as-needed basis.

A *policy* associated with the booster is used to selectively invoke the protocol functions. For example, a forward error correction code might be used over a wireless data link to bring its error behavior into an acceptable operating range, without using the FEC end-to-end [13]. The error performance of the subnet is thus "boosted" to an acceptable level to improve end-to-end performance. Figure 1 shows a booster used in a network, in this case boosting a subnet between an end-host and a router.

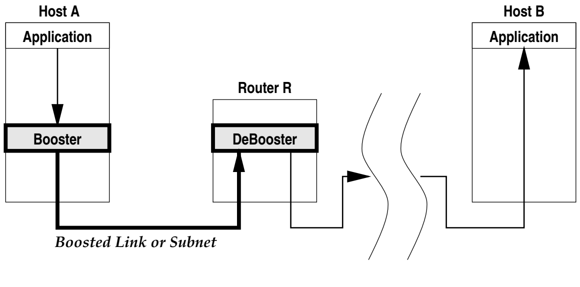

Figure 1: Boosting a link or subnet

Boosters can be dynamically added and deleted as additional network functionality is needed. A policy for this decision is needed in addition to the specific booster mechanism for adding functionality. Since boosters vary widely in their functions, it is impossi-

[1]This research was supported by the Defense Advanced Projects Research Agency under Contract #DABT63-95-C-0073. Additional support was provided by the Hewlett-Packard and Intel Corporations.

ble to have a completely general policy; policies must be associated with their boosters.

These policies can be quite subtle, and may include definition of "meta"-policies. For example, consider two boosters, one that compresses data, and a second that encrypts it. If compression is performed first, the later encipherment of data might in fact be slightly strengthened. However, if encryption is performed first, the compression is unlikely to be effective. A policy module can be devised which properly structures the interaction of these two boosters, for example by indicating that the boosters are not commutative.

1.2 Packet Modification

A *transparent* booster does not modify the packet it boosts. For example, a Forward Error Correction (FEC) booster may send FEC packets *in addition to* the data packets it encodes. Non-transparent boosters, on the other hand, modify data packets. For example, a compression booster for use on a wireless link might compress data packets.

Transparency has architectural implications; non-transparent boosters are *partitioned*; the sender boosts the packet, and a "debooster" at the receiver deboosts and recovers the original packet. This is the situation shown in Figure 1.

1.3 Implementing Boosters

Implementation of boosters requires dynamic insertion of protocol elements into a protocol graph. In practice, protocol graphs are implemented as executable modules that cooperate via messages or shared state. Booster support requires inserting and removing the booster's function from the execution path followed for a group of packets handled by the protocol. A simplified illustration of one style of booster is shown in Figure 2.

While future operating systems[1, 7] may ease user-level implementation of protocols with good support for efficient user/kernel boundary crossing and structured user control of devices[6], today's operating systems are ill-suited for such implementation. Access to system resources needed for high performance, such as address maps and fine-grained scheduling, leads to protocols embedded in operating systems. The canonical example is the IP protocol stack embedded in BSD UNIX. Implementing protocol boosters in this environment allows us to evaluate the technique's applicability today in a realistic setting.

Our overall goal is to show that Protocol Boosters are a good idea. As a first step, we must show that the idea can be realized with acceptable performance. To do this, we implemented several example Protocol Boosters embedded in a BSD TCP/IP implementation, and measured the costs and overheads. We used the FreeBSD implementation of UNIX, operating on Intel Pentium processors interconnected by 10 Mbps Ethernet cards. The availability of freely distributable UNIX sources such as FreeBSD and Linux has made such machines extremely attractive as OS development platforms, and allows free distribution of systems such as the one we have implemented. It is our hope that other boosters and improved OS support will result as others absorb and react to our implementations.

The remainder of this paper is organized as follows. Section 2 motivates particular design choices reflected in the implementation. Section 3 discusses several example Protocol Boosters. Section 4 discusses aspects of the implementation in FreeBSD. Section 5 presents performance data and some inferences we can draw from it. Section 6 discusses related work, and Section 7 concludes the paper with a discussion of lessons learned, new directions and a pointer to the source for our implementation.

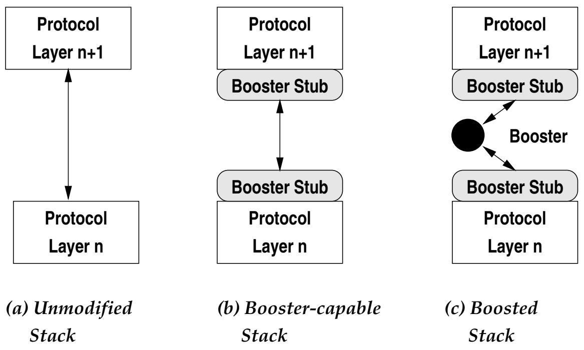

Figure 2: Insertion of Protocol Boosters in a Layered Protocol

2 Implementation choices and strategy

As Figure 2 shows, due to its generality and simplicity, the booster abstraction can be used in many protocol architectures. There is a wide range of implementation alternatives.

2.1 Kernel vs. User level

The initial design choice was whether to run boosters inside the kernel protection domain, or to operate in user-space. Each choice has major consequences for required operating system support.

Running boosters as kernel modules can increase performance, because of context-switching and other overheads, as well as availability of control and information about arriving packets. As many boosters commit layer violations, such information can be very important. Unfortunately, boosters as kernel modules are difficult to debug. Boosters running in user space are much easier to debug, as well as easier to adapt to other operating systems.

Since one role of boosters is as performance-enhancers interoperating with existing network protocols, we implemented prototype support for boosters as kernel modules. This decision should be re-examined as technology advances.

2.2 Platform choice

We added support to FreeBSD, a free Unix clone for the Intel x86 processor architecture. There were two reasons: (1) no cost for a free BSD Unix inspired OS and its source code, and (2) excellent documentation; [18] has an excellent treatment of the BSD networking code. The placement of this implementation in the IP stack is shown in Figure 3.

In our prototype we simplified the policy decision for boosting: all packets destined to (or sourced from) a specific IP address are boosted or de-boosted as necessary. This choice allowed us to investigate the OS performance independent of policy research and development. This is accomplished by a demultiplexing algorithm, as illustrated in Figure 3, which examines the IP address and based on a table lookup, either invokes an appropriate booster or reinserts the packet in the normal execution path. Insertion or deletion of booster functionality is thus controlled by choice of IP address.

2.3 Protocol Layer and implications

A completely general environment for protocol boosters would allow placement at any protocol layer. The key lessons about feasibility and performance of OS

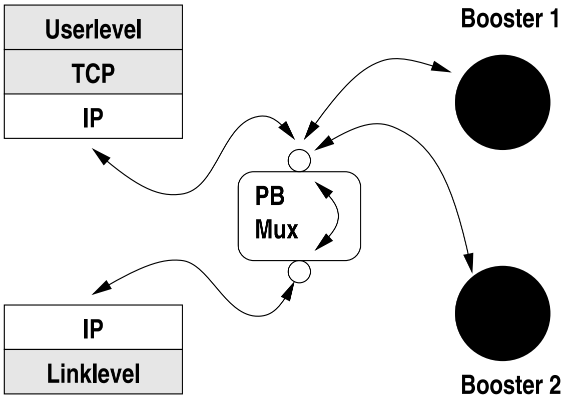

Figure 3: Embedding and selecting boosters in the FreeBSD IP stack

support can be learned with a prototype operating at a single layer. The choice of this protocol layer has important implications for software engineering, limitations of the prototype, and performance measurement. We used the IP layer.

The major software engineering issue other than kernel- vs. user-space placement is interacting with the existing layers. We defer discussion of Software Engineering until Section 4.3, after the implementation details are discussed.

Several limitations were introduced by using the IP layer. These were related to packet fragmentation and reassembly and multipath routing, and are a direct consequence of operating at the IP layer.

Packet fragmentation and reassembly is performed by IP at hosts to avoid the performance cost of repeatedly carrying it out as packets traverse an internetwork. A Maximum Transfer Unit (MTU) is determined for an IP route, which has the property that it requires minimal fragmentation and reassembly. Where a link has a smaller Maximum Transfer Unit (MTU) than the packet size, the packet is fragmented into pieces of MTU size or smaller. The debooster receives the original boosted packet as two (or more) packet fragments. This presents a problem where the booster functionality requires the *entire* original packet. Since this requirement is booster-dependent, our prototype OS implementation by supplies the MTU of the outgoing interface to the booster so it can act appropriately.

Multipath routing occurs since Internet packets are not guaranteed to be delivered, take a particular route, or arrive in-order. TCP addresses the first and third problems as an IP overlay. This IP behavior can present a problem for boosters, especially non-transparent boosters where appropriate deboosters or *state* necessary to deboost the boosted packet are not present. It also complicates inserting and deleting boosters at necessary locations in an IP internetwork. While routes rarely change, as shown by Claffy[3] in her studies of Internet traffic, such routing dynamics can be addressed by future protocol boosters.

The ability to measure performance in a convincing and reproducible manner was our highest priority. Since application performance is an excellent measure of end-to-end performance, measurement of delay and throughput was performed with widely-used tools which measure these parameters using IP protocols and sockets. Thus, we had to implement at the IP layer or below to use these tools. The results suffer in reproducibility if we use subnet specific boosters; the first subnet-independent layer is the IP layer. This argued strongly for an IP-layer implementation; we discuss the specifics of performance measurement in Section 5.

3 Prototype Boosters

We have implemented two example boosters: an encryption booster (for passage across insecure subnets) and a compression booster (for passage across bandwidth-limited subnets). Both boosters have a trivial policy mechanism in which a booster is inserted or removed from the protocol graph by explicit user requests.

3.1 Lempel-Ziv Compression Booster

Lempel-Ziv is a commonly used compression algorithm which finds duplicate strings and replaces the repeating occurrences with a pointer back to the original instance[19]. In the case of limited bandwidth networks, a compression booster might increase end-to-end performance, reducing required throughput, at the cost of increased CPU activity. Compression of various packet components has proven successful for low-bandwidth networks[12].

Placing compression at the network level enables all network services to benefit from compression without any added user-level complexity. Sophisticated policy mechanisms can be put in place with the compression booster to detect the proper conditions for insertion into and removal from the protocol graph.

For instance, a typical problem in congestion detection and avoidance is propagating the network information across a WAN. However, a sophisticated policy/compression module could address congestion somewhat differently and immediately compress network streams based solely on information gathered locally, such as packet loss information used by TCP in making flow control decisions.

3.2 Lucifer, an Encryption Booster

Lucifer is an encryption algorithm developed by IBM in 1971; it was a precursor to the now heavily used DES (Data Encryption Standard) algorithm[21]. In the case of sensitive data traveling over an insecure subnet, an encryption booster can transparently increase the security of the network services provided.

As with many software-based encryption techniques, the performance of the encryption booster as shown in Section 5 is poor due to its CPU-intensive nature. Naturally, encrypting the data with special purpose hardware would improve performance significantly, and this could easily be done with a booster which detects and uses such hardware.

The performance of software-based encryption highlights an important point. For sensitive data traveling between secure clouds, it may be less expensive to encrypt the data only over the insecure hop thereby reducing CPU cost on the endpoints. For example, the boosted link in Figure 1 might be insecure, and the policy module could detect this by destination IP address or other means. Moreover, with the use of special purpose hardware, one could multiplex the hardware across many possible endpoints.

The Lucifer booster is based on widely-available code written by one of us and published in Schneier[17]. Converting this Lucifer code from a user program to a network protocol booster required less than a hour.

4 Implementation in FreeBSD

The majority of OS support as well as the booster modules are loadable kernel modules. The remaining OS support is modifications to the kernel networking code. The modules are loaded with an *ioctl()* system call. Our modified kernel can dynamically load and unload support for protocol boosters.

4.1 Initial BSD network stack

When a datagram arrives at the hardware interface, the hardware puts the datagram into the IP input queue and schedules a software interrupt to execute the IP input routine[18]. This routine processes each datagram on its input queue and returns when the entire queue has been processed. During processing, the IP input routine verifies the IP header checksum, processes IP options and forwards the datagram if necessary. If the datagram has reached its final destination, it is passed to the appropriate higher-level protocol.

On output[18], higher-level protocols like TCP and UDP fill in as much of the datagram as they can, *e.g.*, the TCP header, and then pass the datagram to the IP output queue. This fills in the remaining fields in the IP header, like the checksum, determines the outgoing interface to pass the datagram to, fragments the datagram if necessary and then calls the interface output function.

4.2 Protocol Booster support in the network stack

The basic architecture of our implementation in the IP stack is illustrated in Figure 3. The following subsections explain how it is done.

4.2.1 Identifying boosted packets

In the IP header, the type-of-service is field is not used. We used this field to store the booster id of boosted packets. If boosters need headers or trailers added to packets, they must allocate space and perform the appropriate checksumming themselves.

4.2.2 Input

When a packet arrives at an interface, it is passed to the IP input routine (`ipintr()`) by the hardware. If the packet is destined for another host, it is passed on to `ip_forward()`, which forwards packets appropriately. If the current machine is the final destination, the packet is passed to the protocol booster input routine. At this point, the packet consists of the IP and TCP headers as well as whatever data is in it. The protocol booster (PB) input routine determines whether the packet is boosted or not; if it is not, it returns at once. Else, it passes the packet to the appropriate debooster routine, which attempts to deboost the packet and return the deboosted packet to `ipintr()` to finish processing the packet. Figure 3 illustrates much of this behavior.

4.2.3 Output

A packet can arrive at the IP output routine `ip_output()` in two states : boosted or unboosted. If it is boosted, then the packet is destined for another host, and has been passed to `ip_output()` by `ip_forward()`. If it is not boosted, then it may either be coming from the local host, or it may have also come from `ip_forward()`.

All outgoing packets are passed to the PB output routine. This routine determines whether to boost, deboost or simply forward the packet before sending it out. The packet(s) are then passed back to `ip_output()` which processes and fragments them as appropriate before sending them to the hardware interface.

4.2.4 Booster interface to OS

The interface with the kernel is simple. It can be viewed as consisting of basically two functions - one to boost and the other to deboost. Minimal examples of such functions are given in Figures 4 and 5.

The booster registers these functions by inserting function pointers to them into a lookup-table. This table is then used to demultiplex incoming and outgoing packets. The function called on output of a packet (*i.e.,* the boosting function) is passed 4 parameters - a pointer to the original mbuf chain containing the packet, a pointer to the contiguous memory block into which the packet has been spilled, a pointer to memory that has been allocated to it (the booster) and the MTU of the interface on which the packet is being sent. The deboosting function, usually called when a packet is received, is passed the same parameters except for the MTU size.

Since boosters can generate new packets as well as modifying the old ones, they are required to fill in a data structure that indicates the packet(s) generated. This data structure consists of a linked list of pointers to the start of packets and the length of these packets. We process this linked list to extract the packets and repackage them into mbufs which can then be processed by the rest of the networking code.

4.2.5 Protocol Layer

While our implementation supports boosters at the IP layer, it would be easy to add booster support that

```
int null_boost(struct mbuf *m0,
               char *pcPacketBuffer,
               char *pcPlayground,
         int iMtuOut)
{
   register struct ip *ip = (struct ip *) pcPacketBuffer;

   ip->ip_tos = ip->ip_p;
   ip->ip_p   = IPPROTO_PBOOSTERS + PB_BID_NULL;

   SegmentInfo_pSegmentInfo = (SegmentInfo *) pcPlayground;
   SegmentInfo_pSegmentInfo->pcPacketStart = pcPacketBuffer;
   SegmentInfo_pSegmentInfo->iPacketLength = m0->m_pkthdr.len;
   SegmentInfo_pSegmentInfo->pNextSegment = NULL;
   return (0);
}
```

Figure 4: Example Null Booster for FreeBSD

```
int null_deboost(struct mbuf *m0,
                 char *pcPacketBuffer,
                 char *pcPlayground)

{
   register struct ip *ip = (struct ip *) pcPacketBuffer;

   ip->ip_p = ip->ip_tos;

   SegmentInfo_pSegmentInfo = (SegmentInfo *) pcPlayground;
   SegmentInfo_pSegmentInfo->pcPacketStart = pcPacketBuffer;
   SegmentInfo_pSegmentInfo->iPacketLength = m0->m_pkthdr.len;
   SegmentInfo_pSegmentInfo->pNextSegment = NULL;
   return(0);
}
```

Figure 5: Example DeBooster for FreeBSD

works at the TCP or UDP levels (above IP in the stack), or one at the Ethernet level, which is below IP in the stack.

4.3 Software Engineering Challenges

4.3.1 Interacting with Mbufs

FreeBSD's network information and datagrams are stored and processed in mbufs (memory buffers). Mbufs have a maximum size and are chained into a linked list containing a datagram if the datagram's length exceeds the size of a single mbuf.

We began by passing the packets to the boosters as the mbufs in which they were encapsulated. However, the mbuf structure proved awkward to manipulate, particularly for boosters that operate on contiguous pieces of data, *e.g.*, the compression booster.

We allocate a 32KB memory buffer in which we gather packets as they arrive. A booster is passed a pointer to this contiguous region of memory. We allocate an additional 32K chunk of memory used by the boosters in their processing.

While slightly constrained, implementing each of our sample boosters became very simple. The current FreeBSD kernel network data structures restrict schemes like protocol boosters, Application-specific Safe Handlers[7] and SPIN[1] modules. To exploit ideas from these new systems, the FreeBSD kernel must be made more "extension-friendly".

4.3.2 Kernel-awareness and user-level calls

Implementation issues which are of minor consequence in user space can have devastating side-effects if errors are introduced in the kernel protection domain. Our sample boosters (compression and encryption) were all essentially constructed by simply taking the skeleton algorithm of existing applications (Lempel-Ziv, Lucifer, *etc.*) at the user-level and turning it into the main routine for the booster modules.

The implementations made calls to user-level libraries, or to system calls. Since these are not available in the kernel, we had to implement any required functions. Memory allocation was particularly obscure, so our implementation provides each booster with a pointer to 32K of allocated memory which the booster is expected to manage.

Authors of boosters in our prototype must be 'kernel-aware'. However, it is undesirable for the authors of boosters to completely master FreeBSD internals. More complex boosters will require more powerful and extensive services. A clearly-defined, powerful interface to the kernel should be implemented to provide the most important facilities available to user-level applications; this would greatly accelerate importing existing code into a kernel-resident booster framework.

5 Performance Evaluation

The goal of our performance evaluation experiments was to measure the overhead introduced by our implementation as well as the costs of executing the example boosters. Our experimental setup consisted of two 133 MHz Intel Pentium processors equipped with 32MB of EDO RAM with support for burst reads, a 256KB pipeline write back cache, and 3COM 3c509 ISA Ethernet cards operating at 10 Mbps.

We recorded the roundtrip times of ICMP ECHO (ping) packets of varying sizes between the two hosts, with a number of different boosters installed. This provided an understanding of the delay overhead imposed by boosters, and allowed us to quantify per-byte and per-packet overheads.

We analyzed the throughput of the resulting network stacks using the `netperf` tool[10]. We have experimented with both `ttcp` and `netperf`, and have drawn two conclusions from these experiments. First, `netperf` results are reproducible; `ttcp` measurements exhibit significant variation in reported throughput – up to 20% in some cases. Second, `netperf` results correspond very closely with maximum `ttcp` reported throughputs. What this suggests is that `netperf` better controls the variables under study, while reducing noise from other factors.

5.1 Delay measurements

Figure 6 shows the variation in ping round-trip times with packet sizes ranging from 60 to 1400 bytes and different boosters.

Figure 6 shows that there is virtually no difference in delay between a kernel with booster support enabled and an unmodified FreeBSD kernel. The overhead added by a "null" booster ('spilling' the packet and reassembling it into mbufs) is incurred by all boosters. This overhead is very small, between 0.1 and 0.2 ms, and remains constant with increasing packet size, implying that the cost is per-packet, rather than per-byte.

The Lempel-Ziv booster is much more expensive than the null booster for small packet sizes, but the cost decreases with increasing packet size. We attribute this to increasing compressibility with increase in packet size, so that the increase in processing time is offset by the decrease in the time needed to transmit the data. The Dumb-Lempel-Ziv booster, which compresses the data but sends the original packet rather than the compressed one, behaves as expected - round-trip ping times increase linearly with time, reflecting the processing overhead involved in compressing the packet. It might seem odd that the Dumb-Lempel-Ziv booster ever outperforms the Lempel-Ziv booster. The Lempel-Ziv booster compresses the packet at the source, transmits the compressed packet and decompresses it at its destination, while the Dumb-Lempel-Ziv booster performs the compression computation at the source but sends the uncompressed packet, bypassing decompression at the destination. Therefore the Dumb-Lempel-Ziv booster starts to outperform the Lempel-Ziv booster when the additional time required to de-

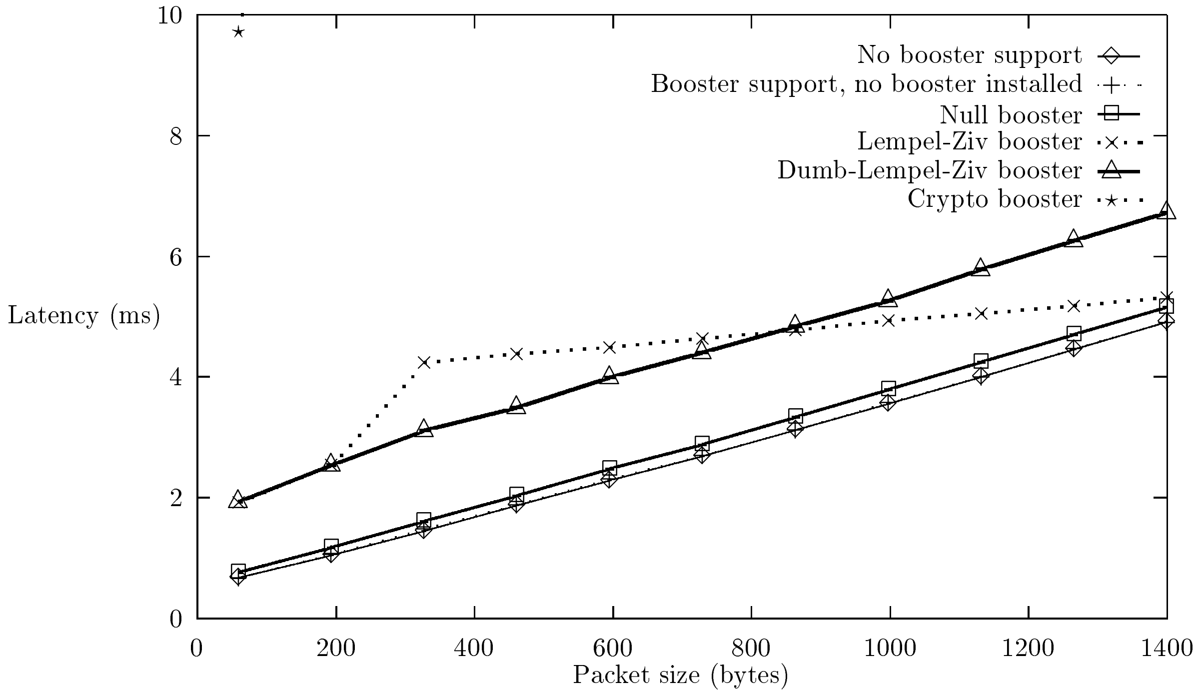

Figure 6: Ping round-trip times

compress the packet exceeds the time gained by transmitting a compressed packet. With increasing packet size and compression gain, this discrepancy lessens until Lempel-Ziv starts to outperform Dumb-Lempel-Ziv.

The roundtrip times of the Encryption booster are very large even for small packet sizes; only the time for the smallest packet is shown in Figure 6 (it is in the upper left corner of the plot). The other times are correspondingly ridiculous.

5.2 Throughput measurements

`Netperf` uses a client-server model to measure the throughput, with one machine acting as the server to the others client. We measured bulk data transfers using TCP and BSD sockets.

The experiments used the test setup described at the beginning of this section, the modified kernel, and no other machines on the Ethernet link. The experiments were repeated until a 99% confidence interval in the results was reached, using an option provided by netperf. The command line used was:

```
netperf -F design.txt -H logos -I 99,5
        -i 10,2 -l 60
```

Table 1 shows the results of our tests. The first column in the table shows the code path being executed, the second column shows the throughputs obtained by netperf, and the third shows the percentage change in throughput relative to a kernel with no booster support installed, which is given as the first row of the table.

The additional processing overhead incurred by our FreeBSD support for protocol boosters has a negligible impact on throughput; with no booster installed, throughput stays the same (7.21MBit/sec). For the null booster, with the associated packet spilling and resegmentation costs, throughput decreases by 0.01%, from 7.21MBit/sec to 7.15MBit/sec.

The measurements also indicates the effect on throughput when boosters which perform significant processing are employed. The Lempel-Ziv booster performs Lempel-Ziv compression, as described in Section 3, and then sends the compressed packet; the Dumb-Lempel-Ziv booster also executes the compression code but sends the original packet, thus incurring all of the cost but none of the benefits of the Lempel-Ziv booster. This provided us with an upper bound on the cost of the Lempel-Ziv booster. The Crypto booster encrypts its data stream using the Lucifer[21] algorithm.

Stack configuration	Throughput (MBit/s)	Percent Change
No support installed	7.21	0.00%
Support installed, no booster	7.22	0.00%
Null booster	7.15	-0.01%
Lempel-Ziv booster	9.42	30.65%
Dumb-Lempel-Ziv booster	5.22	-27.60%
Crypto booster	0.34	-95.28%

Table 1: Netperf statistics

The Lempel-Ziv booster improves performance by up to 30%, approaching the maximum link-level bandwidth when compressible data (such as text files) is being sent. The throughput obtained with the Dumb-Lempel-Ziv booster provides an estimate of the worst-case behavior, decreasing throughput by up to 28%. This is encouraging, implying as it does that even a relatively unsophisticated implementation of the proposed technique for protocol enhancement produces significant performance gains in some fairly common cases.

The Crypto booster, on the other hand, decreases throughput to 5% of its normal value. Since the costs incurred by the OS support are negligible, this decrease in throughput comes from computations for encrypting and decrypting the data stream. Clearly, inefficient or computationally expensive boosters may cause dramatic reductions in throughput.

5.3 Installation costs

Boosters are intended to be added and deleted dynamically to react to network dynamics. In our prototype, this is done from the user level. Inserting the module for booster support into the running kernel takes an average of 30ms (29000 μs). Adding an actual booster takes an average of 20ms (19000 μs). Both of these are times spent executing in the kernel on account of a user-level *ioctl()* request, and exclude concurrency-control costs.

Where a booster's functionality is dynamically inserted and deleted under control of a kernel-resident policy module, the operations can be considerably faster. The simple kernel data structure operations consist of three pointer updates. These can be accomplished while a single processor-priority based lock is held. The cost of lock acquisition and release is less than 100 instructions.

6 Relation to other work

The University of Arizona's x-Kernel[11] work provides support for composing protocols from simpler elements. Protocol boosters are examples of such elements, but they are inserted "on-the-fly". More recent work on the Scout[14] project seeks to use compiler technology to optimize protocol stacks by reducing them to minimal sets of functions. This optimization approach is static, where a general protocol architecture is pared away by optimization technology to achieve a high-performance protocol. Protocol boosting, in contrast, is additive and dynamic; protocol boosters are added when necessary.

Dynamic modification of protocols is not a new idea; for example the notion of building a FILO queue (stack) of reentrant modules is embedded in the UNIX System V STREAMS implementations patterned on Ritchie's Streams[15]. Unfortunately, Streams are restrictive with respect to flow control (they resemble a string of co-routines), module scheduling, and intermodule messaging. Boosters have a smaller set of such restrictions, in fact they are in practice almost unrestricted. This means that the range of protocol architectures which can be implemented is enhanced; for example there are protocol features (*e.g.*, multiplexers) which are difficult to implement with the implicit flow control of the STREAMS message-passing discipline, and easy to implement with boosters.

For example, composition properties are essential for the many-to-one, one-to-many, and many-to-many forms of multiplexing in communications systems. The "waits-for" dependencies used to schedule coroutines would force multiplexers (and demultiplexers) to be single-threaded and data-driven, rather than clock or priority driven.

Protocol boosters have a strong intellectual relation to the application-specific services approach suggested by the University of Washington's SPIN[1] project for building an extensible microkernel. A different tack is followed by MIT's Exokernel[7]; the Exokernel concentrates on allowing applications to specify almost *all* elements of their OS substrate, without focusing specifically on network protocols. Application-specific Safe Handlers (ASHs) are most similar to protocol boosters. A major difference is the focus on protocols in our work; it lets us take advantage of considerable structure inherent in protocols. OS support for protocol boosters occupies a middle ground of generality between STREAMS and an extensible OS.

7 Suggestions for further work and Conclusions

Sophisticated policy modules are clearly essential for many classes of dynamic behavior. Our prototype implementation requires users to explicitly ask for boosters to be inserted and deleted. Automating insertion and deletion of boosters under control of a policy module (*e.g.*, a "compressibility detector") is underway. David Feldmeier[9] has observed that monitoring congestion on a WAN to determine when compression should be applied has desirable properties; compression is then used only when the WAN is congested, and compression reduces throughput, a correct response to WAN congestion.

Our implementation provides access to packets at the IP layer. This was based on our requirements for reproducible measurements, as discussed in Section 2.3. On input, for example, we pass the packet to the booster module after error-checking has been performed; some applications may wish to pass the datagram to the booster module before error-checking.

More general protocol graph support in the FreeBSD would allow adding booster modules at any level in the protocol hierarchy as well as at arbitrary points in the processing of the datagram. This would offer finer-grained control of boosting. For example, one could implement TCP Vegas [2] using protocol boosters if we inserted booster modules at the TCP layer. A desirable implementation target is *x*-Kernel-like protocol graph facility, with access to FreeBSD resources, and with smart policy modules.

To achieve that target, further OS support for boosters should include library routines accessible to booster modules, similar to libraries available at the user-level. This would would insulate programmers from many details such as kernel memory allocation, and let them focus on the algorithms used in the boosters themselves. From a software and protocol engineering perspective, it would save effort, since many boosters have common support needs.

7.1 Summary

Our prototype shows that it is possible to dynamically insert and delete protocol elements in a conventional TCP/IP stack operating under UNIX. Support for these "protocol boosters" can be implemented efficiently; there is a very small performance cost relative to the cost of executing the protocol element's functions. We analyzed these costs using both network throughput and network delay measurements made with widely-available tools; our source code can be obtained via anonymous FTP[2] for those wishing to replicate our measurements or experiment with new boosters.

8 Acknowledgments

The Lempel-Ziv booster is based on publicly-available code by Ross Williams[20]. Tony McAuley and Dave Feldmeier of Bellcore have provided important ideas and commentary. Comments from Scott Nettles greatly improved an earlier draft of this paper.

References

[1] B. Bershad, *et al.*,, "Extensibility, Safety and Performance in the SPIN Operating System," Proc. 15th SOSP, pp. 267–284, December 1995.

[2] L. Brakmo and L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet," in *IEEE Journal on Selected Areas in Communications*, **13(8)**, Oct. 1995, pp. 1465–1480.

[3] K. Claffy, "Internet Traffic Characterization," Ph.D. Thesis, UCSD, 1994.

[4] David D. Clark, Van Jacobson, John Romkey and Howard Salwen, "An Analysis of TCP Processing Overhead," in *IEEE Communications Magazine*, 27(6), June 1989, pp. 23–29.

[2] `ftp.cis.upenn.edu:~ pub/dsl/boosters.d`

[5] D. Clark & D. Tennenhouse, "Architectural Considerations For A New Generation Of Protocols," Proceedings of ACM SIGCOMM, pp. 200–208, September 1990.

[6] P. Druschel, L. L. Peterson and B. S. Davie, "Experiences with a High-Speed Network Adaptor: A Software Perspective," pp. 2–13, Proceedings, 1994 SIGCOMM Conference, London, UK.

[7] D. Engler, *et al.,* "Exokernel: An Operating System Architecture for Application-Level Resource Management," Proc. 15th SOSP, 1995.

[8] D. C. Feldmeier, A. J. Macauley and J. M. Smith, "Protocol Boosters," Technical Report, U. Penn CIS Dept., 1996. See also `http://gump.bellcore.com/~dcf/boosters/homepage.html`.

[9] D. C. Feldmeier, Personal Communication, May 7th, 1996.

[10] Hewlett-Packard, Information Networks Division, "Netperf: A Network Performance Benchmark (Revision 2.0)," Feb. 15, 1995. See also `http://onet1.external.hp.com/netperf/NetperfPage.html`.

[11] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering,* **17(1)**, Jan. 1991, pp. 64–76.

[12] V. Jacobson, "Compressing TCP/IP Headers for Low-Speed Serial Links," Internet RFC 1144, February 1990.

[13] A. J. McAuley, "Error Control for Messaging Applications in a Wireless Environment," INFOCOM 95, Boston, MA, April 2-6, 1995.

[14] A. B. Montz, *et al.,* "Scout: A communications-oriented operating system," Technical Report 94-20, Dept. CS, University of Arizona, June 1994.

[15] D.M. Ritchie, "A Stream Input-Output System", in AT&T Bell Laboratories Technical Journal, October 1984, 63(8) part 2, pp. 1897-1910.

[16] J. H. Saltzer, D. P. Reed, & D. D. Clark, "End-to-end Arguments in System Design," Proceedings of the 2'nd IEEE International Conference on Distributed Computing Systems, pp. 509–512, April 1981.

[17] B. Schneier, "Applied Cryptography: Protocols, Algorithms and Source Code in C," Wiley 1994, pp. 485–491.

[18] W. Richard Stevens and Gary R. Wright, "TCP/IP Illustrated, Vol.2 - The Implementation," Addison-Wesley, 1995.

[19] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory", Vol. 23, No. 3, pp. 337-343.

[20] Nico E. de Vries, "Lossless Datacompression Sources Kit," 1996. Email nevries@aip.nl to obtain a copy.

[21] J. L. Smith, "The Design of Lucifer, A Cryptographic Device for Data Communications," IBM Research Report RC3326, 1971.