Protocol Boosters: Applying Programmability to Network Infrastructures

William S. Marcus, Bellcore
Ilija Hadzic, University of Pennsylvania
Anthony J. McAuley, Bellcore
Jonathan M. Smith, University of Pennsylvania

ABSTRACT This article describes a novel methodology for protocol design, using incremental construction of the protocol from elements called protocol boosters on an as-needed basis. Protocol boosters are an adaptation technique that allows dynamic and efficient protocol customization to heterogeneous environments. By design, the boosting mechanism is under control of a policy, which determines when augmentation is required. Thus, many portions of a protocol stack execute only as necessary, permitting significant increases in performance relative to general-purpose protocols. Design principles for protocol boosters are presented, as well as an example booster. Two implementation platforms are described: (1) an augmented Linux operating system, which is freely available to other researchers; and (2) a rapidly reprogrammable hardware prototype, called the Programmable Protocol Processing Pipeline (P4), which is based on off-the-shelf FPGA technology. Since protocol boosters are programmed functions and can be network-resident, a programmable network infrastructure is necessary to exploit their full capability. Thus, protocol boosters are an ideal application for an on-the-fly programmable network infrastructure.

At the heart of the success and power of the worldwide IP Internet are the general-purpose protocols in the TCP/IP protocol suite. Although these protocols, such as TCP and IP, provide a flexible framework for building diverse applications, two limitations can be seen precisely because of their success and generality. First, they evolve more slowly than the changes in networking technology and application requirements. Second, they trade some loss in efficiency for their ability to handle increased heterogeneity.

In many instances existing protocols do not operate well for emerging applications or take advantage of novel network technologies. Cases in point include TCP's poor handling of the World Wide Web or IP's inability to capitalize on subnetworks which offer quality of service (QoS) guarantees. Both the user community and the Internet Engineering Task Force (IETF) recognize these deficiencies, but remedial action has been slow mainly because of administrative and political barriers rather than technological ones. For example, in 1992 the potential problem of address exhaustion and the degree of options processing in IPv4 led to the three-year design cycle of IPv6. However, three more years have expired and IPv6 has still made insignificant penetration into operational networks. Paradoxically, it is the success of IPv4 that has led to this problem. The problem is not with the specific protocol or standardization method, but with the need to have many people (sometimes with competing agendas) agree on the standard and its deployment. Forming a consensus within large groups is a slow process, and is likely to remain slow; therefore, protocol standards will continue to evolve at a slow pace.

General-purpose protocols are designed to operate in heterogeneous network environments by minimizing the services required from lower layers. Minimizing lower-layer service requirements allows robust operation over the widest variety of network infrastructures, but prevents the protocol from taking full advantage of lower-layer services. Ideally, a protocol would adapt (e.g., be reprogrammed) to provide the best possible performance given the path of the data. For instance, when on a LAN the protocol would adjust itself to give performance similar to that of a specialized LAN protocol. It is this need for protocol adaptation and programmability that has driven our design and implementation of protocol boosters.

PROTOCOL BOOSTERS

Protocol boosters [1] are a methodology for protocol design aimed at overcoming the slow evolution and inefficiencies associated with general-purpose protocols. They incrementally construct protocols from elements called *protocol boosters* on an as-needed basis. Protocol boosters allow:

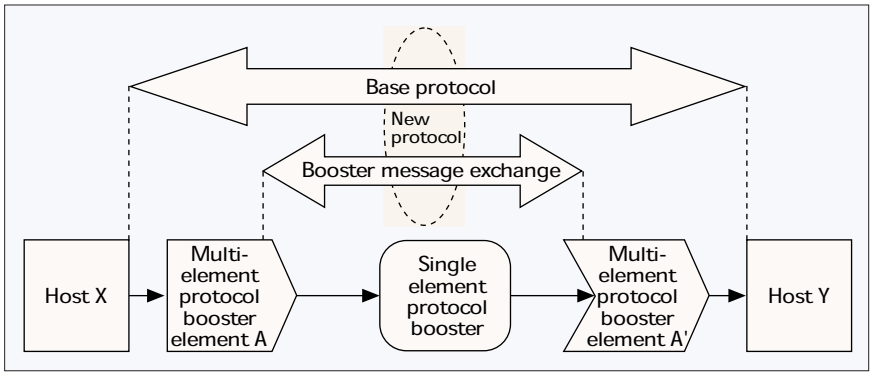
- Design of protocols under optimistic assumptions
- On-the-fly customization of a protocol to heterogeneous environments
- Transparent and rapid deployment, independent of standards, into existing network infrastructures

When applied, this protocol construction methodology improves the network performance and/or augments bearer services. We envision that the protocol booster methodology shall hasten the development and deployment of new protocols for the rapidly expanding collection of network and application technologies.

By definition, a protocol booster adds, deletes, or delays messages of an existing protocol, but does not originate or terminate that protocol. A protocol booster consists of one or more booster elements. The elements of a booster may use protocols to exchange messages among themselves, but these protocols are originated and terminated by the booster elements, and are not visible external to the booster. The definition of a protocol booster can be relaxed to allow message conversion between booster elements in networks which assume static routing (e.g., ATM), but on the periphery of the entire booster, message syntax must be maintained. Consequently, protocol boosters are transparent to the protocol being boosted. Boosters can reside anywhere in the network or end systems and are designed to improve the performance or features of an existing protocol.

Multiple protocol boosters can operate on the same pro-

 0163-6804/98/$10.00



■ **Figure 1.** *The protocol booster model.*

tocol, using either concatenation or nesting of boosters. The transparency of boosters to the underlying existing protocol means that any boosters designed to operate on a protocol can be arbitrarily ordered, although this may not always produce desirable properties, e.g., compression followed by forward error correction. Protocol booster nesting is more sophisticated than concatenation. Although a booster itself is not a protocol, a protocol and a multi-element booster combine to form a new protocol. The new protocol consists of messages of the original protocol combined with messages exchanged among the booster elements. A second booster can be applied to the new protocol in which it is nested. Booster nesting can continue to an arbitrary depth.

With a set of nested boosters, it is natural to talk about a protocol family generated by the boosters. The base protocol is the protocol operated on by the outermost booster of the nested boosters. Additional family members are generated at deeper and deeper levels of the booster nesting. Thus, protocol boosters can be viewed as modular building blocks for a family of related protocols, each suited to a specific network environment.

A policy associated with a booster determines under which conditions booster functions are invoked. Since boosters vary widely in their functions, it is impossible to have a completely generic invocation policy. Policies may be based on, but not limited to, observed network behavior, packet source and destination, or time of day. These policies can be quite subtle and may even include metapolicies aimed at preventing undesirable effects when nesting or concatenating booster elements.

Figure 1 illustrates the above concepts. It depicts boosting communications, via the nesting of a single element booster within a multi-element booster, between two hosts, labeled Host X and Host Y. The establishment of the boosted channel was arranged via a policy decision which determined that the network path between Host X and Host Y required additional support, beyond that which could be provided by the base protocol. As an example, if the two hosts were running an application requiring low latency but were connected via a high-latency, high-loss satellite link, the boosters would conspire, at the expense of additional bandwidth and processing within the network, to hide the retransmission latency from the application. If Host X was running a similar application between itself and another host on its local low-latency subnet, the boosters would not be deployed; thus, the overall protocol processing between the two endpoints would be minimized. Note that the end-to-end communication properties between Host X and Host Y are not compromised. The two hosts are unaware of the protocol booster modules inserted in their communication path; they continue to communicate using the original base protocol. The multi-element booster messages are originated and consumed by the booster elements, never to be processed by the two hosts. Likewise, the single-element booster can monitor the protocol message flow of the augmented base protocol and add, delete, or delay legal protocol messages to or from the message stream.

Practical Example: The FZC Booster

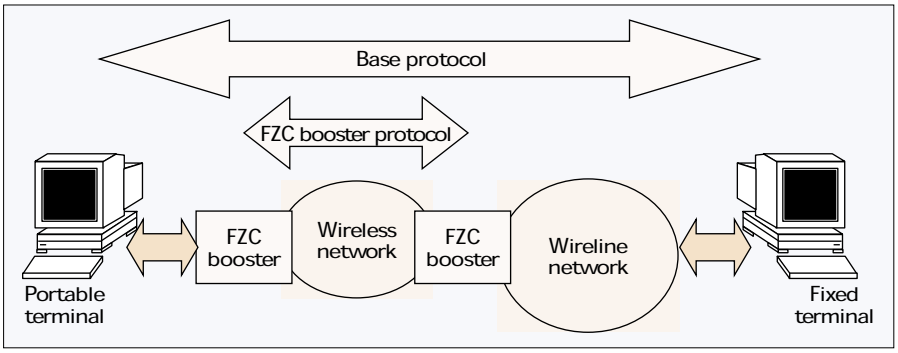
Our Forward EraZure Correction (FZC) booster is a good example of a multi-element booster. It reduces the effective packet loss rate on noisy links, such as terrestrial and satellite wireless networks. Although packet error correction is normally most efficiently and flexibly done by packet retransmission (automatic repeat request, ARQ), forward error correction (FEC) is desirable for some latency-constrained and multicast applications, or where the return channel is unavailable or slow and where the loss of a single packet causes other packets to need retransmission. The FZC booster uses a packet FEC code with erasure decoding. This booster is part of our TCP/IP booster family, which also includes ARQ boosters, a reorder booster, and an error detection booster. Each booster is designed to provide a specific function and work harmoniously with the other boosters in the family. The FZC booster is not well suited for dealing with packet loss due to congestion; other members of our TCP/IP error control booster family handle this situation.

Figure 2 highlights the basic operation of the FZC booster. At the transmitter side of the wireless network the FZC booster adds parity packets. The FZC booster at the receiver side of the wireless network removes the parity packets and regenerates missing data packets. This appears similar to link FEC; however, link FEC only corrects bits (or words), not IP packets or TCP segments, and cannot be applied between any two points in a network (including the end systems). Also, the FZC booster can be applied incrementally. In Fig. 2, for example, we could add an extra FZC booster at the fixed terminal. If this booster adds h1 parity packets and the second booster adds $h2$ parity packets, the portable terminal can recover from up to $h1 + h2$ packet erasures. In the reverse direction the second booster could reduce the number of redundant packets to reduce congestion in the wireline network.

Booster Implementations

We added protocol boosters support to the Linux 2.0.32 operating system for the i386 (Intel) architecture [2]. Kernel-level implementation offers efficiency and transparency. Individual boosters are implemented as loadable kernel modules (lkms), which allow dynamic loading/removal of booster modules into network elements at runtime. Since lkm support is not unique to Linux, this implementation is amenable to other variants of UNIX.

The implementation's system components consist of *booster instances*, *booster sequences*, and *booster traps*. A booster instance is an instantiation of a protocol booster lkm that behaves in accordance with arguments passed to the booster when it is invoked or with arguments passed via ioctl system calls. A booster sequence is a concatenated chain of booster instances, and a booster trap directs individual packets to a specific booster sequence. Our implementation currently limits sequences to 16 booster instances and traps boosted channels by source and destination IP addresses. Stronger trapping based on other information, such as protocol-specific port numbers, can be added.



■ **Figure 2.** *Communications over a wireless network with an FZC booster.*

We implemented the aforementioned FZC booster on this platform. This booster caches, then immediately forwards, each data packet it receives, whether the packet is from an upper-layer protocol or the IP forwarder. The only modification to each data packet is that the FZC booster overwrites the IP packet's 16-bit identification field with a sequence number, allowing the decoder to know the packet's position information. Practically speaking, this does not change the end-to-end UDP datagrams or TCP segments. However, if an application requires, IP options can be used for packet sequence information. After receiving k packets (k is defined per channel) on a given channel, the cached packets are zero-padded to the size of the largest packet in the cache. Also, each packet's size and protocol type are appended to the packet's tail. The transmitter performs an FZC matrix multiplication over the payload, padding, and appended tail of the k packets. The h overcode packet payloads produced by the FEC encoder are then prepended with an IP header and a booster header. This IP header contains a prototype field identifying it as a protocol booster packet and a sequence number in the 16-bit identification field. The booster header contains the type of booster (FZC booster), the value k, and the sequence number of the first of the k packets. The h packets are then transmitted toward the same destination as the data packets.

The FZC debooster at the receive side caches incoming data packets and immediately forwards them either to an upper-layer protocol or toward their eventual destination. Overcode packets are also cached, but are not forwarded. Packets are released from the cache when either:

1. An entire collection of k data packets are present.
2. The received data packets plus parity packets equal k.
3. The cache occupancy dictates cache content replacement.

Only in situation 2 are the matrix computations performed to generate the missing data packets.

To assess the effectiveness of this booster arrangement we deployed it in a simulated wireless environment running UDP. The amount and types of errors on wireless networks depend on link conditions and link error control. In general, however, errors are not random but come in a burst of consecutive bits. Whether the packet errors are also bursty depends on the size of packets and average burst error lengths. Based on the actual packet loss results obtained in satellite experiments, we constructed two basic packet error models: random and bursty. These error models were used in a loss module (implemented as a booster lkm) that can delete packets based on the chosen error model.

As our application we used the public domain Test TCP program (`ttcp`) with the UDP option. We ran `ttcp` between two 166 MHz Pentium-based machines on a dedicated 10base-T Ethernet. Both machines ran the FZC booster, and the receiver also ran the loss module. The experiments used a packet size of 512 bytes, with 1000 packets/trial and a block size of 20 packets. As expected, the results show that with no parity packets the effective packet loss is roughly the same as the loss rate defined in the loss module, and that increasing redundancy decreases the effective packet loss [3]. Although burstiness further increases the network packet loss, the FZC booster is still effective. In fact, larger block sizes make the difference between bursty loss and random loss negligible. In this test environment our implementation was shown to have no performance impact on traffic that is not boosted, and a performance impact commensurate with particular enhancement modules on traffic that is boosted. In particular, comparing the runs of `ttcp` between two hosts having no booster infrastructure present with runs of `ttcp` between the same two hosts having the booster infrastructure installed and a null booster lkm present reveals no perceptible change in throughput. The null booster merely logs packet and byte counts. Replacing the null booster lkm with FZC booster lkms using 4 percent ($k = 50$, $h = 2$) and 30 percent ($k = 20$, $h = 6$) overcode, respectively, reduces the application-to-application throughput to approximately 7.7 Mb/s and 3.9 Mb/s (from 9.6 Mb/s achievable on an Ethernet), which is still very acceptable for most current access network technologies.

Many protocol boosters that we have designed perform bit-intensive operations, and are thus computationally expensive on general-purpose processors. This is a significant potential limitation, since the operating regime where boosting increases end-to-end performance may be very small, and thus designers may be resistant to using the technique, particularly in high-speed backbone networks. An example of this phenomenon is file compression; users now compress only where network access is slow or the file to be transferred is very large.

Protocol boosters are a discipline for use of programmable network infrastructures. Therefore, if, in the design space of programmable network infrastructures, other factors (cost, complexity) could be traded off against performance, the idea of protocol boosters should still be useful. Field programmable gate arrays (FPGAs) offer such a performance trade-off. The first and most important fact is that the devices can now be reprogrammed on the fly; that is, new logic can be downloaded rapidly into devices that are in-system. Thus, the design space of programmable network infrastructures includes hardware. The trade-off is this: there is little or no storage on the devices themselves, and the complexity of boosting operations is severely limited by the densities and sizes of the devices, yet the devices are certainly capable of operating at extremely high data rates.

We have explored this trade-off experimentally with our Programmable Protocol Processing Pipeline (P4) prototype [4] on ATM cells streams at the OC-3c line rate. The P4 channels selected ATM cells (e.g., a virtual circuit) through logic in a series of up to six SRAM-configured FPGAs (Altera 8000 series devices). A crossbar switch performs the channeling of cells to the various devices. Rate decoupling between devices is performed via FIFOs dedicated to each FPGA. The crossbar can be reconfigured in approximately 1 µs. If some protocol-processing element is necessary, the hardware implementation (i.e., FPGA configuration) can be downloaded into the FPGA device in approximately 100 ms on an as-needed basis. Unnecessary modules are simply removed from the protocol stack by disconnecting the FPGA device from the processing stream and putting it back into the pool of available empty devices. The freed device can

later be downloaded with some other configuration and used for some other processing.

We have demonstrated the operation of an FEC encoder and decoder running at the OC-3 link rate (155 Mb/s) and boosting the TCP/IP (over ATM) protocol stack in a noisy environment by dynamically adding and removing FEC [5]. The FEC consists of an $R = 1/2$ convolutional encoder and a Viterbi decoder. The encoder consumes one FPGA device, the decoder four devices. To assess the effectiveness of this booster arrangement we constructed a testbed consisting of a 300 MHz Pentium II-based machine running Linux 2.0.29 with a Fore PCA200 ATM NIC, two P-4 systems, an ATM switch, and a broadband network analyzer. `ttcp` tests were performed using the FEC booster running at the OC-3 rate with a range of bit error rates (BERs) spanning several orders of magnitude (from 10^{-12} to 10^{-4}). For BER > 10^{-7} (e.g., a bad link) TCP benefits from the FEC booster, while it represents a performance cost for BER < 10^{-7} (e.g., a good link). At BER around 10^{-5} an improvement of a factor of 10 has been observed, and the improvement grows exponentially as the BER becomes worse. At BER 10^{-4} and worse, TCP without FEC completely stalls, while TCP with FEC is still able to operate.

Protocol Boosters, Active Networks, and Other Adaptation Techniques

Programmable network infrastructures such as *active networks* [6] have attracted attention recently, particularly due to their potential for accelerating network evolution. Network evolution occurs when the network changes to benefit applications with advanced services. With programmable networks, once the service is defined the network can be customized to optimize for the delivery of the service. Simple examples include Web proxy caches, specialized firewalls, and adaptive FECs.

Protocol boosters represent a design methodology for network protocols. The design methodology is centered on the use of transparent, composable protocol functions called "protocol boosters" which are injected into protocol stacks at hosts and inside the network. This latter role relies on the presence of a programmable network infrastructure. Thus, one simple view of protocol boosters is an application of active networking.

A second, and more compelling, view of protocol boosters is as a design methodology for programming active networks. Several design decisions which underlie the current definition of protocol boosters were motivated by our experiments with early boosters. An example is the design restriction that a booster can operate correctly (albeit with lower performance) in the absence of any booster-aware code at a receiver. This seems like a rather odd restriction, but it is strongly motivated by the preference of packet-switched networks for dynamic rerouting. Table 1 illustrates where boosters are robust.

If we presumed that routes were static, we could assume that a boosted packet or message would encounter a debooster simply by traveling along a path where we had placed one. However, given that dynamic routing provides fault-tolerance advantages, it is clear that choosing a model where no debooster is required opens the approach to a wider range of networks. Thus, protocol boosters can be profitably viewed as a programming style for active networks. While active networks are intended to provide flexible programming of networks, this flexibility will not result in robust systems unless disciplined.

Protocol boosters also share a similar problem space with:

- Link-layer adaptation
- Protocol termination [7]

Static routing	Dynamic routing	
x	x	No debooster required
x		Debooster required

Table 1. *Booster robustness in static and dynamic routing.*

- Protocol conversion [8]
- Special end-to-end protocols [9]

Link-layer protocol adaptation operates independent of the higher-layer protocols. This greatly simplifies link protocol design, but can have many undesirable side effects, such as increased latency on applications that do not require the adaptation or uncomplementary protocol interaction with those that do. Protocol termination is the serial concatenation of several protocols, each tailored for their local environment, operating along the path of an application's data channel. While protocol termination permits the exchange of information between two or more entities over heterogeneous networks, it very often comes at the cost of losing desirable end-to-end properties. In addition, protocol termination exposes applications to single points of failure and longer per-packet processing times. Protocol conversion is very much like protocol termination, with the restriction that only the protocol's syntax is changed; thus, the desirable end-to-end properties are maintained. Like protocol termination, protocol conversion exposes applications to single points of failure and longer per-packet processing times. Protocol boosters have the advantage over these two techniques in that applications are not further exposed to single points of failure. Like these two other solutions, protocol boosters do, in most cases, induce increased packet processing times to the applications being boosted. Special end-to-end protocols remove the overhead associated with general-purpose end-to-end protocols. This solution creates protocols that are specifically designed with a single application and network infrastructure in mind. The performance of such protocols is unmatched by any of the other alternatives; however, this solution suffers from a long and expensive development time which may not be cost-effective for niche applications.

Summary

We described a method for protocol design based on incremental, dynamic construction of protocols on an as-needed basis. The elements of these protocols, protocol boosters, can be composed in a "Tinkertoy"-like manner, to form a family of protocols. The methodology motivates optimistic protocol design, where protocols are designed assuming the best case, and adds additional functionality on an as-needed basis. This is in direct contrast with building for the worst case and optimizing by finding common fast paths through the protocol implementation. Prototypes of the booster design methodology have been implemented. The performance was sufficiently encouraging that a larger-scale design is being investigated.

Protocol boosters can be viewed as a step toward the fully programmable infrastructure proposed by a number of researchers under the rubric of active networks. While many of the problems are the same (e.g., robust end-to-end behavior, security for systems into which boosters are loaded, etc.), a key advantage of boosters is that they can easily be injected into today's systems without a wholesale change in network infrastructure. In that sense, they offer an early test of the promise, as well as a programming model, of active networks.

The protocol booster methodology offers some exciting possibilities for accelerating the evolution of protocols,