Efficient Cross-Layer Negotiation

Bryan Ford
Max Planck Institute for Software Systems*
baford@mpi-sws.org

Janardhan Iyengar
Franklin & Marshall College
jiyengar@fandm.edu

ABSTRACT

Internet evolution often depends on either inserting new protocol layers or upgrading existing layers to new protocols, but both of these evolutionary paths are obstructed by the difficulty and inefficiency of determining *which* protocols a pair of hosts mutually support and prefer. We propose a novel cross-layer Negotiation Protocol that sets up a complete stack of connection-oriented protocols at once, concurrently performing handshaking for multiple layers and choosing among alternative protocols for each layer in as few round trips as possible, often just one. The initiator proposes a protocol graph explicitly encoding possible configurations along with protocol-specific handshake data; the peers then prune, refine, and atomically commit to a final configuration, exchanging messages over a specialized transport that can operate in-line with the negotiated protocol stack. Although a practical Negotiation Protocol presents many challenges, our initial exploration suggests that these challenges are solvable, and we believe addressing them is a necessary step toward a more evolvable Internet.

1. INTRODUCTION

The Internet's power and generality rests on its architectural use of layering [35], which enables diverse transports and applications to run atop IP, and IP in turn to run atop diverse physical networks [6]. Internet extensions often insert new layers, such as cryptographic security [8, 18] and addressing enhancements [22, 23]. Resolving architectural conflicts created by middleboxes [1, 13, 15] may entail further decomposing the Transport Layer [12], and interim solutions already create deep layer cakes, such as "IPsec-on-IPv6-on-HTTP-on-TLS-on-TCP-on-IPv4" in Microsoft's DirectAccess [7]. A key missing ingredient, however, is a mechanism to *decide* efficiently which protocols implementing which layers to use between a given pair of hosts. This absence is hampering the Internet's ability to evolve effectively, either by introducing new layers or by upgrading existing ones cleanly.

More layers can increase processing costs on end hosts, and methods of mitigating these costs are well-studied [4, 5, 16]. Layering connection-oriented protocols can also increase connection setup delay, however. With TLS [8] atop TCP [33], for example, each protocol's setup requires at least one round trip, and TLS's handshake cannot begin until TCP's completes. More connection-oriented layers stacked this way accumulate more setup delay, which can quickly become noticeable when latency is high. Since long-distance round trip delay is limited by the speed of light, and does not decrease with technological advances, the cumulative round trip costs of connection setup may be the most important cost of layering in the long term.

Keeping new layers—and new implementations of old layers—interoperable with legacy hosts also has costs. If an application wishes to run atop SCTP [32] but fall back to TCP for compatibility, for example, the application cannot start its TCP handshake until it receives an ICMP Protocol Unreachable [27] response to its SCTP attempt—or until SCTP times out, in the common event an intervening middlebox silently drops SCTP packets. Speculatively opening alternative connections in parallel wastes host and network resources, and these costs compound with additional alternatives. A DNS extension could allow querying supported protocols, but such "out-of-band" negotiation incurs the administrative cost of updating the DNS server whenever the host's software stack changes, and fails if a middlebox on the path blocks a protocol that both endpoints support. Even within a single protocol such as TCP, version evolution becomes simpler and cleaner given some negotiation meta-protocol [24].

Revitalizing the Internet's evolution requires a systematic solution to these layering challenges. To this end we propose a novel cross-layer *Negotiation Protocol*, which can concurrently negotiate and initialize a complete stack of connection-oriented protocols between a pair of hosts in as few round trips as technically feasible. The key idea underlying the Negotiation Protocol is for end hosts to express a set of *possible* protocol stack configurations they support, along with protocol-specific handshaking data, in an explicit *protocol graph* encoded in one message. A connection initiator first sends an initial protocol graph "proposal," which the responder modifies by "pruning" unsupported or undesired alternatives, and possibly elaborating the graph further before returning it to the initiator. The participants refine this protocol graph while speculatively processing the initialization handshakes of all protocols present in the graph at each stage, until all alternatives are decided and the hosts commit to a configuration. As the example exchange in Figure 1 illustrates, negotiating any number of layers and alternatives often requires only one "3-way handshake."

*at Yale University as of September 2009

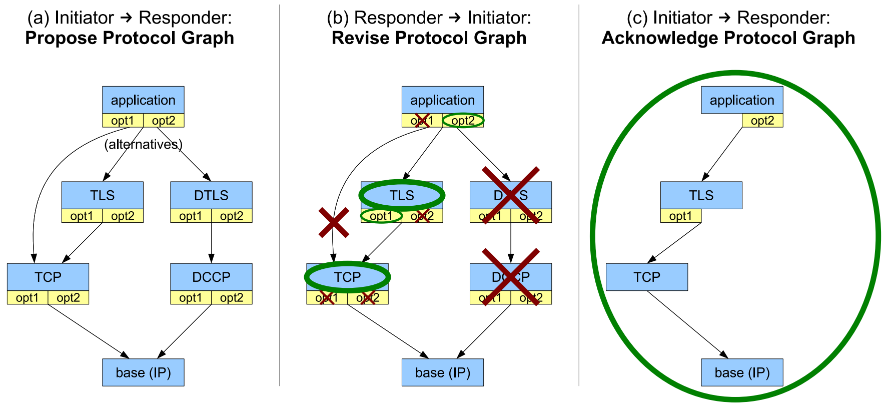
(a) Initiator → Responder:
Propose Protocol Graph
application
opt1
opt2
(alternatives)
TLS
DTLS
TCP
DCCP
base (IP)
(b) Responder → Initiator:
Revise Protocol Graph
(c) Initiator → Responder:
Acknowledge Protocol Graph

Figure 1: Example Negotiation Protocol exchange constructing a two-layer protocol stack (transport layer plus security layer) in one "3-way handshake."

Designing a practical Negotiation Protocol offers several technical challenges, such as efficiently encoding the protocol graph and protocol-specific data, packetization and congestion control of large negotiation messages, throttling of "speculative" communication on behalf of protocols that may not be included in the final protocol graph, handling situations in which parts of the negotiation process need to be secured (authenticated and/or privacy protected), and allowing negotiation to fall back gracefully to legacy protocols when a communication peer does not support the Negotiation Protocol. We explore these challenges and tentative solutions in the context of a prototype under development.

Section 2 presents our conceptual model for negotiation, independent of protocol design details. Section 3 then describes our prototype Negotiation Protocol. Section 4 outlines interactions between the Negotiation Protocol and other system components. Section 5 discusses outstanding issues, Section 6 outlines related work, and Section 7 concludes.

2. A MODEL FOR NEGOTIATION

This section describes our conceptual model for cross-layer negotiation, independent of negotiation protocol details, which we defer to the next section.

2.1 Protocol Graphs

Inspired by modular networking architectures such as the x-Kernel [16, 25], we model a running protocol stack as a directed acyclic graph whose nodes represent *protocol instances* such as TCP connections, and whose directed edges represent *dependencies* or "builds-upon" relationships between these instances. For example, a TLS session running atop a TCP session has an edge from the TLS node to the TCP node. If a communicating system's "steady state" is conceptually a graph of protocol instances represented by state in the end hosts, the Negotiation Protocol's purpose is to build and initialize that state efficiently based on the capabilities and requirements of the communicating hosts.

Since negotiation involves not only setting up protocols but agreeing on which alternative protocols and options to use, the Negotiation Protocol's graph represents a union of *possible* protocol stack configurations. For example, if an application can run atop either TCP or SCTP, the protocol graph contains nodes for both alternatives, and one alternative is "pruned" during negotiation. Individual protocols may negotiate protocol-specific options at the same time, such as TCP's SACK [20] and time stamp [17] options, using the protocol's existing header and option formats. The Negotiation Protocol is thus a "container" carrying handshake data for several protocols at once.

In the example protocol graph in Figure 1(a), the application can run directly atop TCP, atop TLS over TCP (for security), or atop DTLS over DCCP (for secure datagram-oriented communication). Each protocol node is annotated with any options and parameters specific to that protocol. This graph structure can represent a wide range of configuration alternatives compactly: in a simplistic "list of complete protocol suites" representation, for example, the TCP node and its handshake option data would be duplicated in the alternatives with and without TLS.

2.2 Basic Negotiation

In a basic exchange, the connection initiator *proposes* a protocol graph describing configurations supported by the initiator, and the responder replies with a corresponding graph from which it has *pruned* unsupported or undesired alternatives, leaving the representation of a "definite" protocol stack from which the hosts set up connection state. In Figure 1(b), the responder selects TLS over TCP by pruning all other graph nodes, and selects certain protocol options but prunes others. In (c), the initiator acknowledges this negotiated graph and communication commences using the configured protocols.

As long as the initiator can (and wishes to) express its full configuration possibility space in the first message, and all protocol-specific handshakes require only one round trip, this basic exchange takes only one "3-way handshake" regardless of protocol graph complexity.

For DDoS protection, the responder typically does not set up long-term state at step (b) but returns a cookie along with the revised protocol graph, which it later verifies upon receiving the initiator's acknowledgment (c) before initializing its state. The responder may however maintain some quota of "cookie-free connections," so that if the responder is lightly loaded and a cookie-free slot is available, the responder sets up its state immediately without returning a cookie challenge. In this case, useful communication may commence *during* the first round trip, e.g., using handshake data embedded in the application protocol's graph node.

2.3 Extended Negotiation Scenarios

While we expect many common cases to fit the basic pattern above, some situations may require additional graph revision steps possibly involving more round-trips. We point out several such negotiation scenarios here.

Deferred Graph Elaboration.
If the initiator does not wish to reveal its entire feasible protocol configuration space in its first negotiation message, e.g., due to security or message size concerns, it may replace protocol nodes or entire graph subcomponents with "placeholders" in the initial message, allow the responder to start pruning the incomplete graph in the first round trip, and then elaborate the graph further by filling in these placeholders in subsequent round trips.

Initiator Choice of Alternatives.
The initiator may wish to retain "the power of choice": instead of allowing the responder to choose among alternatives proposed by the initiator, the initiator may insert placeholder nodes for the *responder* to elaborate with protocols and options the responder supports, among which the initiator chooses by pruning the graph a half round trip later. When and what types of placeholders are allowed depends on the rules of the protocol(s) they represent: some protocols may require explicit representation in the first message, others may require that the initiator *always* sends an initial placeholder that the responder fills in with its available alternatives, etc.

Protocol-Specific Information Dependencies.
If protocol B runs atop protocol A and B's protocol-specific handshake process requires information produced by A's handshake process, the protocols are inherently serialized and B's handshake must wait until A's completes. The Negotiation Protocol still completes the exchange with the minimum latency possible given these dependencies, however.

Peer-to-Peer Negotiation.
While the basic negotiation exchange assumes the hosts have asymmetric "initiator" and "responder" roles, this negotiation model might be extended to peer-to-peer situations in which both hosts act as "initiators," e.g., as in a TCP simultaneous open [33] or a NAT traversal rendezvous [10]. In this case, each host independently creates and sends an initial protocol graph proposal; upon receipt of its peer's proposal, each host combines the two proposals to create a final (or next stage) protocol graph. In this case the two peers must use a *convergent* graph pruning algorithm: not only pruning nodes that either peer doesn't support, but making the *same* choices among alternatives they do support, ensuring convergence to the same final protocol stack. Not all protocols need support simultaneous initiation, and those that do must specify choice rules that ensure convergence.

Recursive Negotiation.
An instance of the Negotiation Protocol may need to run recursively "within" another instance: e.g., an "outer" instance may negotiate cryptographic security protocols and parameters in cleartext, while an "inner" instance provides security-protected negotiation of higher-level protocol configuration and parameters. With care, these nested negotiation exchanges can run concurrently and incur no additional round trips. If the initiator knows or has cached the responder's public encryption key, for example, the initiator can encrypt the first message of the inner (secured) Negotiation Protocol exchange, sign it with its own private key, and embed it in the security protocol's handshake data in the outer (cleartext) Negotiation Protocol exchange. If the initiator includes its own public encryption or Diffie-Hellman key in this first message, the responder can similarly embed its first inner (secured) response message within its first outer (cleartext) response, and so on. Such "zero-delay" nesting of Negotiation Protocol instances may require the initiator's security protocol to know or guess an initial encryption algorithm that will be acceptable to the responder (or else embed multiple encrypted versions of the inner request within the outer request, which may be expensive). If the initiator guesses wrong, the responder can simply ignore the embedded inner message and force the initiator to re-encrypt and re-send the inner message in the next round trip.

3. PROTOCOL DESIGN

We now examine our prototype Negotiation Protocol design in detail. We do not claim this design to be the "right" or "best" one, but merely use it to solidify the concepts discussed above and make an initial attempt to identify and address important technical challenges facing a practical Negotiation Protocol.

3.1 Negotiation Context

We assume that any negotiation exchange takes place in some well-defined *context*, which uniquely identifies the communication endpoints and the raw delivery channel atop which negotiation is to occur. For example, if

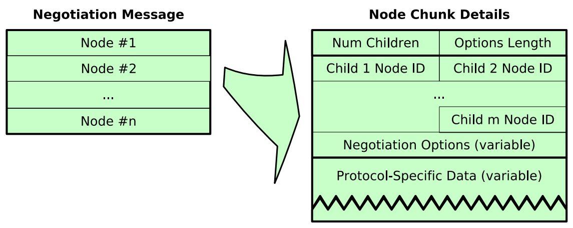

Figure 2: Negotiation Message Structure

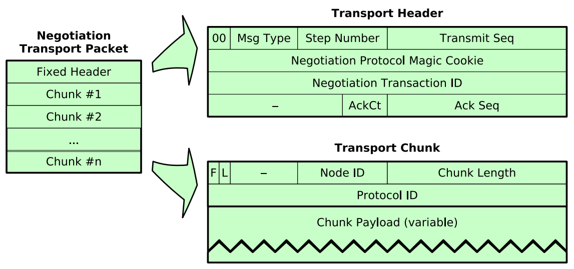

Figure 3: Transport Header and Chunk Format

a pair of hosts wish to run the Negotiation Protocol directly atop IP, then the context is defined by the two hosts' IP addresses and an IP protocol number, and Negotiation Protocol packets sit directly atop the IP header. If two application endpoints wish to negotiate a user-level protocol stack via UDP sockets, then the negotiation context is defined by the UDP session 5-tuple of IP addresses, IP protocol, and UDP port numbers.

Negotiation may be *in-band* or *out-of-band*. After an in-band negotiation, the negotiated protocol stack uses the same communication context as the negotiation itself: e.g., transmits packets over the same UDP session as the negotiation packets. Protocol stacks negotiated in-band must make their packets distinguishable from Negotiation Protocol packets, as discussed below in Section 3.3.

After out-of-band negotiation, the resulting stack uses some other context, such as endpoints agreed upon via protocol-specific handshake data. Out-of-band negotiation is consistent with architectures like NUTSS [14], in which "control plane" and "data plane" communication follow different routes. Out-of-band negotiation in today's Internet however carries the risk that information learned via one path may not apply to the other: the endpoints might find only after negotiation has completed and committed that the chosen data path is blocked by a middlebox. Similarly, peer-to-peer NAT traversal typically requires in-band control signaling [10], leading us to focus here on in-band negotiation.

3.2 Encoding the Graph

Each protocol node in the graph has a *Node ID*, which is fixed within a negotiation exchange but otherwise arbitrary. Each node also has a *Protocol ID*, a well-known value indicating a specific (version of a) standardized or experimental protocol defining the node's meaning.

A negotiation exchange proceeds in one or more *steps*, each party sending one message per step before waiting for the other party's next message. A message is a sequence of *node descriptors*, each describing one protocol graph node, as illustrated in Figure 2. A node descriptor consists of zero or more *child node IDs* pointing to other nodes that (may) build on this protocol; a type-length-value (TLV) field containing optional per-node parameters defined by the Negotiation Protocol; and a payload area containing protocol-specific handshake data.

The first node in the message represents the negotiation context atop which the entire negotiated protocol stack will build, such as IP or UDP. Node descriptor ordering is otherwise arbitrary, allowing hosts to optimize common cases. A host may for example first transmit all nodes comprising the preferred protocol stack, followed by other alternatives; the receiver may begin the next negotiation step as soon as "enough" of the previous step's message has arrived, as described below.

3.3 Negotiation Message Transport

Since a negotiation message could substantially exceed the path MTU if it includes protocol-specific objects such as cryptographic certificates, a special-purpose transport illustrated in Figure 3 packetizes negotiation messages. The negotiation transport outlined here is designed for in-band negotiation in a best-effort delivery context such as IP or UDP; other contexts might entail modifications.

Inspired by SCTP [32], a negotiation transport packet consists of a fixed header followed by one or more *chunks*, each chunk carrying data for a different graph node, allowing several small node descriptors to share one packet.

The header includes a magic cookie following the convention set by STUN [28], allowing in-band negotiation packets to intermix safely with those of other conforming protocols: particularly with those of the eventually negotiated protocol, but also with NAT traversal packets [10] atop peer-to-peer UDP sessions. The header's *Transaction ID* uniquely identifies a negotiation exchange initiated by a given host, and the *Step Number* indicates the relevant step within that exchange.

The sender assigns consecutive transmit sequence numbers to a message's packets so the receiver can reconstruct their original order. Node descriptors too large for one packet receive consecutive sequence numbers; flags demark the first (F) and last (L) chunks of a node, allowing the receiver to reconstruct each node independently even if packets for prior nodes are still missing. *Every* chunk for a given node contains the node's Protocol ID, so the receiver can simply acknowledge and drop all packets for protocols it does not support without ever storing them.

The receiver acknowledges packets by transmit sequence number, enabling the sender to retransmit individual packets, and congestion control the sending of large messages.

Borrowing from SST [11], each packet acknowledges a single contiguous, limited range of sequence numbers, providing the benefits of selective acknowledgments without the complexity of variable-length SACK headers.

4. INTERFACING WITH PROTOCOLS

The Negotiation Protocol could in theory be deployed with no changes to the protocols being negotiated: an application could simply run the Negotiation Protocol on a "raw" graph of alternatives whose nodes contain little or no protocol-specific data, and once negotiation completes, use the pruned protocol graph as a "plan" from which to set up instances of the negotiated protocols using those protocols' normal setup mechanisms. This approach might still derive benefit from selecting among alternatives efficiently, but the setup of the negotiated protocols would require additional round trips, which might be serialized as described in Section 1. For maximum efficiency, therefore, negotiated protocols must be modified to interact with the Negotiation Protocol. This section outlines the modifications and interfaces involved.

The Controller and Graph Setup.

We assume some entity within an end host, which we call the *Controller*, is ultimately responsible for deciding to run the Negotiation Protocol in some context and determining the set of protocol configurations to allow. If an application uses the Negotiation Protocol to build a protocol stack from the transport "all the way up," the application is the Controller. A system-level component may alternatively use the Negotiation Protocol to supervise the deployment of protocols below the networking API, transparently to the application, such as by silently substituting TCP with compatible but more efficient transports like SST, or even with multi-layer suites [12]. In this case the operating system serves as the Controller.

To prepare for negotiation, the Controller first creates a Negotiation Protocol instance for the appropriate communication context, then requests that each of the relevant protocols *register* itself with the negotiation instance to form the nodes of the possible protocol graph. The Controller similarly invokes the protocols to create graph edges representing relationships between protocol nodes.

Negotiating as Initiator or Responder.

When the Controller invokes the Negotiation Protocol to initiate a negotiation exchange, the Negotiation Protocol constructs the explicit encoding of the potential protocol graph, in the process invoking each protocol in the graph to obtain that protocol's initial handshake data and embed it in the graph: e.g., TCP's SYN header including any relevant TCP options. The Negotiation Protocol then sends this encoded graph to the responder.

As packets representing this graph arrive at the responder, the responding Negotiation Protocol reconstructs the node descriptors and handshake data for supported protocols while discarding nodes for unsupported protocols, as described above in Section 3.3. Once a node is complete and its dependencies processed, the Negotiation Protocol invokes the node's protocol, passing the initiator's handshake data, to advance that node to the next handshaking step. The protocol returns the handshake data to be sent back to the initiator: e.g., SYN-ACK data and options for TCP. The protocol may not allocate long-term state outside the protocol graph until the Negotiation Protocol commits. Both the Controller and individual protocols may prune graph nodes deemed less desirable or nonfunctional, such as due to errors in received handshake data.

The responder then sends the revised graph back to the initiator, whose Controller either continues the back-and-forth or commits if the graph has reached a suitable state. Upon commit, the Negotiation Protocol sends an acknowledgment of the final graph to the peer (usually the responder), enabling the peer to commit as well. The Negotiation Protocol then invokes each protocol remaining in the graph to set up its long-term state based on that protocol's last handshake data. At this point the protocol may explore the final graph and interact with other protocols to optimize subsequent communication, e.g., by precomputing offsets and setting up data handling pipelines [6]. The Negotiation Protocol then becomes passive for the rest of the communication session, except to retransmit lost packets of the final commit message as necessary.

5. DISCUSSION

Designing and deploying a practical Negotiation Protocol will require further exploring several important issues, of which we briefly discuss two: backward compatibility and implementation complexity.

Backward Compatibility.

Since today's Internet has no Negotiation Protocol, deploying one incrementally may require enhancing existing protocols such as TCP to use the Negotiation Protocol if both hosts support it, but fall back to the "raw" original protocol if not. This problem may demand protocol-specific solutions. For TCP, an upgraded host could transmit an initial Negotiation Protocol packet, followed immediately by a conventional TCP SYN containing the Negotiation Transaction ID in a TCP option. If the remote host understands the Negotiation Protocol and receives the negotiation packet first, it suppresses subsequently received TCP SYNs from the source containing the matching Negotiation Transaction ID; otherwise, the hosts fall back to legacy TCP operation. (Suppressing SYNs purely by source address and port may be risky due to NATs.)

Implementation Complexity.

Does a Negotiation Protocol's benefits justify its implementation complexity? Our model's basic protocol graph structure and high-level message format are quite simple; much of the complexity in Section 3 resides in the negotiation transport, which may be (or might be made) similar enough to a "workhorse" transport like SCTP or SST to reuse much of the implementation. Economics may also justify this complexity: web site owners may wish for site-wide SSL to protect against prevalent HTML injection attacks [29], and cheap many-core processors and crypto acceleration [34] can mitigate SSL's computational costs, but only cross-layer negotiation in some form will address the risk that SSL's extra round trips will make the site take twice as long as its competitors to appear.

6. RELATED WORK

Modular network architectures have been studied for decades, such as the OSI layering model [35], the Internet [6], the x-Kernel [16, 25], and the Click router [21]. Previous work on optimizing layered protocol stacks has focused primarily on the processing costs to end hosts, as addressed by integrated layer processing [5] and protocol compilation [4] for example. Our Negotiation Protocol complements this prior work by addressing instead the round trip costs of multi-layer negotiation and setup.

TCP [33] uses header options to negotiate extensions, achieving backward compatibility within one protocol at the cost of adding progressively more complexity, overhead, and unpredictability to TCP header processing as options accumulate [17, 20]. O'Malley and Peterson argue against this practice and in favor of simply switching to new "fixed" TCP header formats as needed, via some negotiation process [24]. We agree with the latter philosophy, but feel that a solution is needed for negotiation *across* as well as within layers. Meta-protocols like TCPMUX [19] and the RPC port mapper [31] offer naming indirection but do not negotiate protocol configuration.

Ad hoc enhancements reduce round trip setup costs for specific existing protocols. T/TCP [2] allows closed TCP sessions to be recycled quickly, and persistent connections and pipelining in HTTP 1.1 [9] avoids TCP's 3-way handshake costs on successive requests [26]. TCP [33] originally permitted piggybacking application data onto SYN packets, a feature unsupported by most current TCP stacks due to DDoS and other concerns. DCCP [33] also allows piggybacking, less problematically due to DCCP's best-effort semantics. SST [11] can open lightweight TCP-like streams with no 3-way handshake, after establishing shared underlying association state between the two hosts. The Bundle protocol [30] minimizes round trips across DTNs at application level. These techniques, while effective within a given protocol, do not address the broader need to negotiate efficiently across multiple layers and alternative protocols implementing each layer.

Protocols like RSVP [3] use explicit network layer negotiation to guarantee quality of service along a path. Our focus in contrast is primarily on end-to-end negotiation of transport and higher level services.

7. CONCLUSION

The evolvability of the Internet's layered architecture is currently limited by the difficulty and inefficiency of deploying new layers and new implementations of existing layers. A cross-layer Negotiation Protocol could not only allow hosts to negotiate efficiently among a wide variety of alternative protocol stacks in one round trip, but also piggyback the handshaking of each protocol in the negotiated stack onto that same round trip in common cases.

8. REFERENCES

[1] J. Border et al. Performance enhancing proxies intended to mitigate link-related degradations, June 2001. RFC 3135.

[2] R. Braden. T/TCP – TCP extensions for transactions, July 1994. RFC 1644.

[3] R. Braden, Ed. Resource reservation protocol (RSVP) – version 1 functional specification, September 1997. RFC 2205.

[4] Claude Castelluccia and Walid Dabbous. Generating efficient protocol code from an abstract specification. In *SIGCOMM*, August 1996.

[5] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[6] David D. Clark. The design philosophy of the DARPA Internet protocols. In *SIGCOMM*, August 1988.

[7] Joseph Davies. DirectAccess and the thin edge network. *Microsoft TechNet Magazine*, May 2009.

[8] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, August 2008. RFC 5246.

[9] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.

[10] Bryan Ford. Peer-to-peer communication across network address translators. In *USENIX*, April 2005.

[11] Bryan Ford. Structured streams: a new transport abstraction. In *SIGCOMM*, August 2007.

[12] Bryan Ford and Janardhan Iyengar. Breaking up the transport logjam. In *HotNets-VII*, October 2008.

[13] N. Freed. Behavior of and requirements for Internet firewalls, October 2000. RFC 2979.

[14] Saikat Guha and Paul Francis. An End-Middle-End Approach to Connection Establishment. In *SIGCOMM 2007*, August 2007.

[15] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator, January 2001. RFC 3027.

[16] Norman C. Hutchinson and Larry L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[17] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance, May 1992. RFC 1323.

[18] S. Kent and K. Seo. Security architecture for the Internet protocol, December 2005. RFC 4301.

[19] M. Lottor. TCP port service multiplexer (TCPMUX), November 1988. RFC 1078.

[20] M. Mathis, J. Mahdav, S. Floyd, and A. Romanow. TCP selective acknowledgment options, October 1996. RFC 2018.

[21] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *17th SOSP*, December 1999.

[22] R. Moskowitz and P. Nikander. Host identity protocol (HIP) architecture, May 2006. RFC 4423.

[23] E. Nordmark and M. Bagnulo. Shim6: Level 3 multihoming shim protocol for ipv6, June 2009. RFC 5533.

[24] S. O'Malley and L. Peterson. TCP extensions considered harmful, October 1991. RFC 1263.

[25] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *TOCS*, 10(2):110–143, May 1992.

[26] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28(1–2):25–35, December 1995.

[27] J. Postel. Internet control message protocol, September 1981. RFC 792.

[28] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session traversal utilities for NAT (STUN), October 2008. RFC 5389.

[29] Seth Schoen. Detecting packet injection: A guide to observing packet spoofing by ISPs. Electronic Frontier Foundation whitepaper, November 2007.

[30] K. Scott and S. Burleigh. Bundle protocol specification, November 2007. RFC 5050.

[31] R. Srinivasan. Binding protocols for ONC RPC version 2, August 1995. RFC 1833.

[32] R. Stewart, ed. Stream control transmission protocol, September 2007. RFC 4960.

[33] Transmission control protocol, September 1981. RFC 793.

[34] VIA Technologies, Inc. VIA PadLock security engine, August 2005. Technology Brief.

[35] Hubert Zimmermann. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.