Channel-based Unidirectional Stream Protocol (CUSP)

Wesley W. Terpstra Christof Leng Max Lehn Alejandro Buchmann
Databases and Distributed Systems, TU Darmstadt, Germany
{terpstra,cleng,max_lehn,buchmann}@dvs.tu-darmstadt.de

***Abstract*—This paper presents a novel transport protocol, CUSP, specifically designed with complex and dynamic network applications in mind. Peer-to-peer applications benefit in particular, as their requirements are met by neither UDP nor TCP. While other modern transports like SCTP or SST have also tried to combine the advantages of TCP and UDP, CUSP overcomes their technical and conceptual shortcomings.**

CUSP makes it possible to directly express application logic in the message flow. Modern applications need a mixture of request-response, request-multiple-response, publish-subscribe, and message-passing. All of these operations can be conveniently implemented using CUSP's unidirectional streams.

We separate low-level packet management from streams into reusable channels. A channel connects two applications providing negotiation, congestion control, and cryptography. Developers operate on the stream level, sending messages as reliable and ordered byte-streams. Although they may share a common channel, a stall or loss in one stream does not block the others.

I. Introduction

CUSP is a reliable and secure general purpose transport designed with peer-to-peer (P2P) networking in mind. While many transport protocols have been proposed in the past, we believe ours is the first systematically designed to address the specific requirements of P2P applications. We designed CUSP for P2P partly because we believe that P2P covers the requirements of many modern network applications. P2P applications exhibit asynchronous, dynamic, and complex interactions with a large number of communication partners. We believe that many (if not most) existing Internet applications would have benefited from a protocol like CUSP during their design.

Building upon ideas from SST [1] and SCTP [2], CUSP divides the transport into channels which are responsible for low-level packet management and streams which are multiplexed inside of channels. The stream interface allows application designers to directly express application logic in the message flow. Streams are cheap, created without a round trip and thus need not be used sparingly. As not all messages expect an immediate or direct answer, streams in CUSP are unidirectional; bidirectional streams are modelled on top of this primitive. Applications prioritize streams individually, allowing high priority streams to cut in line.

CUSP is implemented on top of UDP making it easy to deploy and reuse established NAT traversal mechanisms [3]. The protocol also offers mobile networking, seamlessly renegotiating channels and resuming streams. The channel layer has built-in cryptography; assured authenticity simplifies its design and cryptographic negotiation is streamlined into channel creation. Though feature-rich, CUSP is a simple protocol and can be implemented in comparably few lines of code.

Traditionally an application designer had to pick either UDP or TCP. Yet, neither fits modern applications. P2P applications especially have several important new requirements also applicable with varying degree to other application domains.

A. Connection-oriented

In the past, P2P overlays aiming for low latency used unreliable UDP datagrams. The main advantage is that UDP has no connection handshake. Of course, it consequently does not have reliable exactly-once delivery, congestion control, authenticity, or privacy. If an application requires any of the above, and thus a round-trip handshake, it might as well use CUSP to get them all. In this respect, CUSP competes with DCCP, TCP, SST, and SCTP, all of which require at least one round-trip to setup and provide only a subset of its features.

CUSP negotiates exactly-once delivery and cryptography in a single round-trip (Section III-B), the minimum possible. Within an established channel, CUSP creates new streams without a round-trip, leveraging the channel state to ensure exactly-once delivery semantics. Congestion control is managed at the channel-level. Thus, new streams wait neither for slow-start nor after packet loss from inter-stream competition.

B. Complex Exchanges

Traditional network applications are usually modelled as simple request-response message exchanges between two hosts. This does not fit P2P for several reasons. First, communication is often indirect. A node issues a query into the overlay and receives replies from several hosts with which it has not communicated directly. Second, messages are often passed from one node to another without a reply (e.g. in a recursively routed scenario). And finally, one message might cause several independent reactions from one host.

When an application uses CUSP, each network action should use a new stream. Therefore CUSP streams must have low message overhead, a small memory footprint, and zero setup delay. In fact it is possible to create a stream, send data, and close the stream all within a single packet (Section IV). As not every action has exactly one reaction, streams are unidirectional. If no reaction is anticipated, a reply message which must itself be acknowledged is both useless and wasteful.

Instead, an action expecting one or more answers supplies in its stream the ID of one reply service for each response type. A complex conversation with many possible forks and outcomes is then modelled using one stream per step in the logical flow. In the case of indirect multi-hop scenarios, reply service IDs can be simply forwarded to the eventual responder.

C. Priorities

P2P systems exchange a wide spectrum of message types, some more time-sensitive than others. A bulk download is lower priority than a user query, which is in turn lower priority than topology maintenance messages. Furthermore, priorities must not only be respected inter-stream, but also inter-channel.

Priorities cannot be implemented properly on top of an existing reliable transport protocol. Losing a segment from one stream will stall all other streams until that segment has been retransmitted. This is called head-of-line blocking. Furthermore, reliable protocols typically have internal buffering, adding latency to any new high priority stream.

In CUSP, every stream has its own associated priority. The highest priority ready stream always sends first; ready streams are those not blocked by flow or congestion control with data to send. Equal priority streams are serviced round-robin.

D. Many Connections

P2P systems establish hundreds of concurrent connections with extremely diverse lifetimes. For example, the BitTorrent [4] client Vuze (formerly Azureus) can quickly reach its default of 200 connections when downloading a popular file. DHTs like Kademlia [5] can easily exceed 500 connections even in a 10,000 node network [6].

For some operations, P2P systems establish very short-lived connections. For example, in unstructured search systems [7] a query is routed through the overlay network. When a peer with an answer receives the query, it connects directly to the source. This connection is then only used to reliably deliver the answer. Since P2P applications constantly open and close connections, we cannot waste state on closed connections.

Competing protocols retain state after a connection has been closed. An actively closed TCP end-point must enter a TIME_WAIT state for four minutes in order to ensure reliable delivery of the last ack. Negotiation protocols like Just Fast Keying (JFK) [8] must cache entire response messages for previously established connections. That means one packet per successful connection for a minimum of four minutes. For P2P networks with many one-shot connections, this is unacceptable. In CUSP, neither streams nor channels retain state after completion. The negotiation protocol (Section III-B) does not require a JFK-style response cache (we solve SYN-flooding differently) and TIME_WAIT is usually avoided at the cost of an additional message (Section III-C).

II. Related Work

The standard reliable transport TCP does not offer anything that CUSP cannot do equally well or better. In contrast, UDP is only a thin layer over IP. It is extremely flexible, but lacks features many applications end up reinventing. We position CUSP as a complete replacement for TCP also appropriate for some applications built on UDP that implement their own handshake for reliability.

DCCP [9] is a connection-oriented protocol adding congestion control to unreliable datagram delivery. Because it is connection-oriented, it sacrifices one of the main advantages of UDP, the zero setup time. If a three-way handshake is performed anyway, then encryption, authenticity, and exactly-once delivery should be at least optional features. DCCP lacks all of them and also has a TIME_WAIT problem like TCP.

SCTP [2] was designed to overcome TCP's weaknesses. Data is transmitted in form of messages whose boundaries are preserved. Like CUSP, it supports multiple streams within one connection, avoiding head-of-line blocking. But the protocol does not allow dynamic stream creation; the number of streams is determined during connection setup. This severely limits its usefulness for complex and dynamic applications. Even though confidentiality, authenticity, and priority management are not part of the SCTP standard, its implementation seems significantly more complex than CUSP's.

CUSP is similar to SST [1], the protocol which inspired its development. We borrow the concept of channels (with congestion control) and streams (with flow control). In SST streams are prioritized and bidirectional; new streams are created by spawning from existing streams. As discussed in Section I-B, unidirectional streams are preferred for complex applications, because they allow more flexible response flow and can be substantially cheaper. SST also suffers from several technical shortcomings overcome by CUSP. First, SST's stream identifiers can fall out of sync, a problem aggravated by unidirectional streams. Second, SST uses JFK which retains state after a channel closes. Third, it enters TIME_WAIT after channel teardown. Fourth, its flow control cannot exclude buffer overruns. Finally, carrying data in negotiation packets violates exactly-once delivery and must be sent in the clear.

CUSP takes the best ideas from SCTP and SST and fits a wider spectrum of interaction patterns than either of them. It integrates confidentiality and authenticity as integral parts of the protocol. Additionally, the resource use of the protocol is tightly controlled. Nevertheless, CUSP is conceptually simpler and easier to implement than both of its progenitors.

III. The Channel Protocol

Communication in CUSP is application-application, not host-host. Applications are identified by their application key, a public key unique to that application on that host. As CUSP is layered on top of UDP, applications contact each other using UDP addresses. This initial contact creates a channel logically associated to the pair of local and remote UDP addresses. However, streams created on this channel are associated not to the remote UDP address, but rather to the remote application key. If the remote application changes addresses (perhaps switching from Ethernet to Wi-Fi), a new channel is automatically negotiated and streams continue unaffected.

The channel layer is responsible for all the low-level packet details. It reports to the stream layer which segments have been lost, limits the transmission rate using congestion control, and negotiates cryptography acceptable to both sides. The channel layer is not responsible for retransmitting lost segments; in fact CUSP never retransmits packets. Rather, the stream layer puts lost segments back on a ready queue, to be transmitted inside a new packet when the stream has priority.

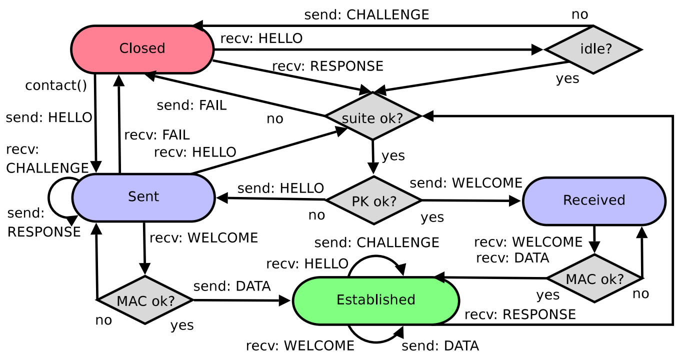


Fig. 1. CUSP negotiation flow chart

A. Normal Operation

During normal operation, DATA packets carrying stream payload are exchanged. Each has a unique sequence number, with the low 28 bits contained in the transmit sequence number (TSN) field. Every established packet ends with a message authentication code (MAC) which ensures the packet belongs to the channel and has not been corrupted. Any packet with an invalid MAC is discarded without further processing.

Sequence numbers start at 0 and increase every packet. In principle sequence numbers never wrap around; they have unbounded length. In practice they are used as nonces for encryption and must not wrap within the nonce space. Before this happens, a channel is disconnected to trigger renegotiation. CUSP only uses nonced MACs; thus, they confirm sequence numbers are correctly extrapolated from the TSN field.

Every data packet acknowledges a contiguous range of sequence numbers. A packet x is acknowledged if $a - l <= x <= a$, where the acknowledge sequence number field (ASN) contains the low 28 bits of a and l is the acknowledge length field (AL). This has most of the advantages of TCP selective acknowledgements (SACK) [10] without a per-packet bitmask.

Currently, CUSP implements a variant of NewReno [11] for congestion control. To simulate the fast retransmit rule [12], we declare an unacknowledged packet x dead if we receive an ASN three packets ahead of x which does not acknowledge x. Section V experimentally confirms this is TCP-friendly.

B. Negotiation

The negotiation protocol is responsible for selecting acceptable cryptography and synchronizing communication. Figure 1 shows the packet exchange sequence is somewhat similar to TCP. The initiator sends HELLO, receives WELCOME, and then sends a (possibly empty) DATA packet. If overloaded, the responder can CHALLENGE instead of WELCOME.

Compared to JFK [8] and TCP SYN cookies [13], our explicit CHALLENGE/RESPONSE defense uses an extra round-trip. However, this defense is implemented so that it need never be used; the CUSP nuclear option. Recall that a successfully completed channel must retain no state. Furthermore, the negotiation must establish a shared secret. To defend against resource attacks and compute a secure shared secret, one must either retain state for failed connections (vanilla TCP), retain state for successfully completed connections (JFK), or spend an extra round-trip. CUSP takes the approach of paying state for the first few failed connections and then switching to spending an extra round-trip. We use a variation of the puzzles from [14] for CHALLENGE/RESPONSE.

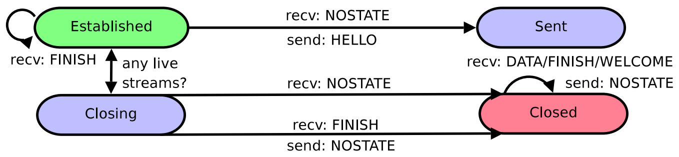


Fig. 2. CUSP tear-down flow chart

Cryptography in CUSP is broken into two suites: the public-key and symmetric suites. A public-key suite includes a Diffie-Hellman (DH) function and a compressor function to hash two DH group elements. A symmetric suite includes a cipher and MAC function. We currently implement only Curve25519 [15] and Whirlpool [16] for the public-key suite and AES128-CTR and Poly1305 [17] for the symmetric suite.

The HELLO and WELCOME packets list cryptographic suites acceptable to an application. Suites have an associated computational cost, and the cheapest mutually acceptable suite is always selected. To reduce the number of round trips, HELLO packets speculatively include DH public keys for the lowest numbered locally acceptable public-key suite. Upon receipt of a HELLO packet, the locally acceptable suites are intersected (bitwise AND'ed) with the remotely acceptable suites. If there is no mutually acceptable public-key or symmetric suite, then the responder reports negotiation FAILure. Otherwise, it checks if the speculatively included public keys are acceptable. If they are, the responder immediately sends a WELCOME packet. Otherwise it sends its own HELLO with the suite intersections and mutually acceptable public-keys.

The WELCOME and DATA packets are cryptographically MAC'ed using the mutually acceptable symmetric suite. This confirms the identity of the remote application before transition to the Established state allows data to flow. Furthermore, it completely eliminates the need for two separate ports in protocols like http/s, imap/s, smtp/s. It is impossible for a man-in-the-middle to pretend that the responder does not want encryption, because the WELCOME packet is authenticated.

CUSP's cryptographic negotiation is based on a 3-message variant of HMQV [18]. This is a very fast negotiation protocol with the guarantees we want. HELLO packets include the initiator's public-key g^a and an ephemeral public-key g^x. The ephemeral key is necessary to ensure forward-secrecy. The WELCOME packet includes the responder's public-key g^b and g^y. The shared secret σ is calculated using the HMQV equations and used to seed the symmetric suite. Each participant generates his sending key as $H(\sigma||g^a)$ and his receiving key as $H(\sigma||g^b)$ where $||$ is concatentation, H is the compressor function, and g^a, g^b are his and his partner's public-keys respectively. We negotiate two different secret keys, because the initiator and responder should use different cipher streams. This construction also tidily deals with simultaneous open.

C. Tear Down

Channels are automatically destroyed as soon as both applications agree there are no live streams. This is conceptually achieved using a Closing state (Figure 2). A channel transitions freely between the Closing and Established state depending on whether or not there are any live streams.

Consider the packet exchange for the last stream. Receipt of stream-level completion moves the receiver into the Closing state, so that the acknowledgement will now have the FINISH type. The sender processes the acknowledgement, clearing the state for the last stream and thus also entering the Closing state. The type of the acknowledgement (FINISH) is now processed, moving the sender into the Closed state and causing the transmission of a NOSTATE packet. The receiver sees the NOSTATE packet and also transitions to Closed. For a single request-response style connection, the packet flow is: HELLO, WELCOME, DATA, DATA, FINISH, NOSTATE.

Compared to TCP, CUSP requires one additional packet, a trade-off that mitigates the TIME_WAIT problem. Upon receiving a NOSTATE packet, we transition from Closing to Closed. Only if the NOSTATE packet is lost will CUSP wait for a 4-minute timeout to trigger the transition. As packet loss is quite uncommon, TIME_WAIT is almost always eliminated.

NOSTATE packets echo back the MAC of whatever packet triggers them. This defends against an attack where a third-party injects a forged NOSTATE packet. CUSP will only honour a NOSTATE packet with a MAC identical to the most recently sent packet. If the remote application really has no state, then congestion control will eventually throttle the channel to the point where there is only a single packet in-flight and the NOSTATE packet will be honoured.

IV. The Stream Protocol

Once a remote application has been contacted and its public-key validated, streams can be created. CUSP does not promise when a stream will be sent; it might be blocked indefinitely by a high priority stream. It only guarantees that when an application is informed that stream shutdown is complete, the remote application has acknowledged the stream.

A. Normal Operation

The stream layer described in this section is layered on top of the channel layer and requires the channel layer guarantee:

- the integrity of received messages
- received messages are never 16 or more TSNs old
- to report possible message loss or confirmed delivery

The stream-layer groups streams by remote applications, as identified by their public-key. Of the streams with pending data unblocked by flow control, the highest priority is reported on a ready-to-send (RTS) signal line. If a channel is not blocked by congestion control, it sets its RTS line equal to the RTS of the attached stream group. To create a packet, the channel layer selects the highest priority channel and provides the stream-layer with a packet to fill. The stream-layer fills this packet with messages from streams in decreasing priority order until either the packet is full or there are no more RTS streams.

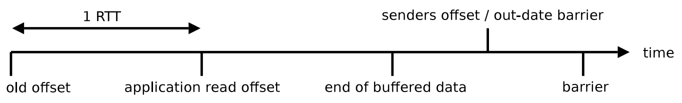


Fig. 3. Flow control with application limited rate

CUSP must ensure that retransmissions will not confuse the receiver into creating the same stream twice or completing a new stream from a retransmitted packet. As CUSP can pack stream creation, payload, and completion into a single packet, the entire stream could be retransmitted if reported lost. A stream identifier is not re-used until the packet creating it and its acknowledgement have been transmitted. Since the channel layer enforces a 16 sequence number age limit, CUSP simply includes a 4-bit monotonically increasing re-use counter with stream identifiers.

B. Flow Control

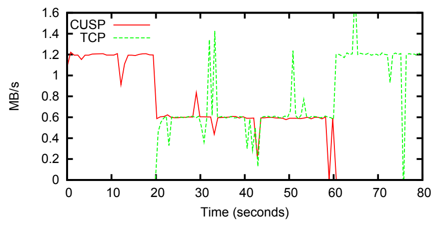
A stream sender may only transmit segments up to a receiver-specified barrier. As the receiver application processes the stream, it moves the barrier forward, allowing the sender to transmit further. The barrier is only ever increased by the receiver. This flow control mechanism guarantees that the sender cannot overrun the buffer space provided by the receiver. If for some reason the sender violates the protocol and exceeds the barrier, a receiver should treat the entire packet as lost. As specified, the policy for barrier updates is implemented entirely by the receiver. This section describes a barrier update policy which performs well in combination with NewReno.

For a new stream, both sender and receiver set the barrier to the minimum window size, 16 times the ethernet MSS. We call the offset up to which the receiving application has read the read offset (RO). The receiver's barrier is the read offset plus the window size. Whenever the receiver's barrier exceeds the last transmitted barrier by more than $\frac{1}{4}$th of the minimum window size, it transmits a BARRIER update to the sender. If this is lost, it transmits a fresh BARRIER update.

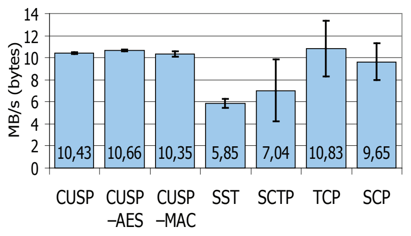
To perform well, this policy must estimate a good window size. To this end, whenever a BARRIER update is acknowledged, the receiver calculates the progress of the RO since the barrier was sent. This is how much the application processed in one round-trip. The window is set to twice this value or the minimum window size. The application processing rate might be limited by the application, flow control, or congestion control. We examine the policy for each of these cases.

If the processing rate is limited by the application, then Figure 3 illustrates the flow control. An entire round-trip's worth of data is buffered locally, with the barrier allowing up to another round-trip's worth of data. If a packet is lost, this buffer provides enough time for fast retransmission to fill the hole before it would stall the application.

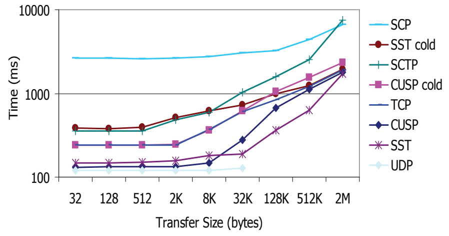
If the processing rate is limited by flow control, the receiver has nearly empty buffers; the application has consumed all the data. In this case, the policy allows exponential growth. Each round trip, the window is doubled, until either the application's processing rate or congestion control become the bottleneck.



(a) NewReno Fairness



(b) Throughput on 100Mb/s Internet Link



(c) Response Times on Intercontinental Link

Fig. 4. Experimental Evaluation

If the processing rate is limited by congestion control, there will likewise be no local buffering. For NewReno's slow start to find the correct congestion window, it must be able to increase the transmission rate exponentially. Thus, flow control must also allow exponential rate increase. Unfortunately, this means that during congestion avoidance the flow control window is bigger than necessary; a stall in the receiver application could cause buffering of twice the congestion window. However, breaking the channel-stream abstraction to improve this one case does not seem worthwhile.

V. Evaluation

We evaluated our implementation (available at [19]) and compared it to UDP, TCP, and SCTP from Linux 2.6.26.

A congestion-controlled protocol should share bandwidth fairly with TCP. In Figure 4a we started a full-rate CUSP stream, consuming all available Wi-Fi bandwidth. 20 seconds later a TCP connection joins in; CUSP and TCP share with perfect fairness. At 40 seconds an additional CUSP stream is added to the channel. This does not affect the bandwidth sharing. When CUSP stops, TCP quickly consumes all bandwidth.

To measure actual network performance we transfer 100MB between two hosts connected to the Internet backbone with Fast Ethernet; the RTT is 5.5ms. CUSP is able to keep up with TCP, even outperforming SCP. SCTP falls behind, as does SST. AES and MAC do not slow down CUSP in practice.

To gauge raw performance, we also measured it on loopback. When AES encryption is negotiated, our Opteron 2.3GHz got 46MB/s (bytes), compared to 103MB/s without. Replacing the MAC with an optimized CRC32 gains 8% more.

We conclude that our user-space prototype performs surprisingly well, especially when compared to its unencrypted siblings SCTP and SST. The throughput tests confirm that an efficient MAC does not significantly impact performance on a real network. Thus it's reasonable to always use a MAC.

As discussed in Section I-B, it is important that stream creation is fast in order to model application flow. In Figure 4, a cold start refers to stream creation without an existing channel. Otherwise, SST and CUSP response times are assumed to already have a channel available. On real networks, CUSP cold start response times match TCP. This isn't surprising as both require an initial handshake, and CUSP's HMQV calculation only takes 0.7ms. SCTP lags behind here due to its four-way handshake. Naturally, a cold start is far worse than both UDP and a hot start which require no round-trips. For small messages, SST and CUSP hot starts are competitive with UDP.

Once the stream size increases beyond the initial receiver window, SST and CUSP must block, waiting for the receiver to increase the window. However, CUSP has a small initial window compared to the fixed 64KB in SST. This gives SST a superficial initial advantage on high latency links where it can push more data before blocking. Once streams become large (like the 100MB bandwidth tests) CUSP's window scaling plays to its advantage and it overtakes SST's throughput.

VI. Conclusion

Designed for the challenging P2P environment, CUSP is a versatile transport with the potential to replace DCCP, SCTP, and SST. It offers a wide range of features from unidirectional streams to encryption. it fits many application scenarios. Our userland prototype outperforms even kernel-level SCTP.

References

[1] B. Ford, "Structured Streams: a New Transport Abstraction," in *SIGCOMM*, 2007.
[2] R. Stewart, "Stream Control Transmission Protocol," RFC 4960, 2007.
[3] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "TCP Extensions for Long-Delay Paths," RFC 5389, Oct. 2008.
[4] D. Qiu and R. Srikant, "Modeling and Performance Analysis of BitTorrent-like Peer-to-Peer Networks," in *SIGCOMM*, 2004.
[5] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *IPTPS*, 2001.
[6] K. Graffi, K. Pussep, S. Kaune, A. Kovacevic, N. Liebau, and R. Steinmetz, "Overlay Bandwidth Management: Scheduling and Active Queue Management of Overlay Flows," in *LCN*, Oct 2007.
[7] W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann, "BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search," in *SIGCOMM*, Aug. 2007.
[8] W. Aiello, S. M. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. D. Keromytis, and O. Reingold, "Just Fast Keying: Key Agreement in a Hostile Internet," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 2, 2004.
[9] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC 4340, Mar. 2006.
[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018, Oct. 1996.
[11] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581, Apr. 1999.
[12] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782, Apr. 2004.
[13] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations," RFC 4987, Aug. 2007.
[14] T. Aura, P. Nikander, and J. Leiwo, "DOS-Resistant Authentication with Client Puzzles," in *SPW*. Springer, 2001.
[15] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," in *PKC*. Springer, 2006.
[16] P. S. Barreto and V. Rijmen, "The Whirlpool Hashing Function," in *NESSIE Workshop*, Nov 2000.
[17] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *FSE*, vol. 3557. Springer, 2005.
[18] H. Krawczyk, "HMQV: A High-Performance Secure Diffie-Hellman Protocol," in *CRYPTO*, 2005.
[19] "CUSP," http://www.dvs.tu-darmstadt.de/research/cusp/, Oct. 2009.