Recent Advances in Reliable Transport Protocols*

Costin Raiciu, Janardhan Iyengar, Olivier Bonaventure

Abstract

Transport protocols play a critical role in today's Internet. This chapter first looks at the recent of the reliable transport protocols. It then explains the growing impact of middleboxes on the evolvability of these protocols. Two recent protocol extensions, Multipath TCP and Minion, which were both designed to extend the current Transport Layer in the Internet are then described.

1 Introduction

The first computer networks often used ad-hoc and proprietary protocols to interconnect different hosts. During the 1970s and 1980s, the architecture of many of these networks evolved towards a layered architecture. The two most popular ones are the seven-layer OSI reference model [119] and the five-layer Internet architecture [27]. In these architectures, the transport layer plays a key role. It enables applications to reliably exchange data. A transport protocol can be characterized by the service that it provides to the upper layer (usually the application). Several transport services have been defined:

- a connectionless service
- a connection-oriented bytestream service
- a connection-oriented message-oriented service
- a message-oriented request-response service
- an unreliable delivery service for multimedia applications

The connectionless service is the simplest service that can be provided by a transport layer protocol. The User Datagram Protocol (UDP) [87] is an example of a protocol that provides this service.

Over the years, the connection-oriented bytestream service has proven to be the transport layer service used by most applications. This service is currently provided by the Transmission Control Protocol (TCP) [89] in the Internet. TCP is the dominant transport protocol in today's Internet, but other protocols have provided similar services [60].

Several transport protocols have been designed to support multimedia applications. The Real-Time Transport protocol (RTP) [101], provides many features required by multimedia applications. Some of the functions provided by RTP are part of the transport layer while others correspond to the presentation layer of the OSI reference model. The Datagram Congestion Control Protocol (DCCP) [67] is another protocol that provides functions suitable for applications that do not require a fully reliable service.

[*] Parts of the text in this chapter have appeared in the following publications by the same author(s): [17], [117], [94], [78] and [61].

C. Raiciu, J. Iyengar, O. Bonaventure, "Recent Advances in Reliable Transport Protocols", in H. Haddadi, O. Bonaventure (Eds.), *Recent Advances in Networking*, (2013), pp. 59-106. Licensed under a CC-BY-SA Creative Commons license.

The rest of this chapter is organized as follows. We first describe the main services provided by the transport layer in section 2. This will enable us to look back at the evolution of the reliable Internet transport protocols during the past decades. Section 3 then describes the organization of today's Internet, the important role played by various types of middleboxes, and the constraints that these middleboxes impose of the evolution of transport protocols. Finally, we describe the design of two recent TCP extensions, both of which evolve the transport layer of the Internet while remaining backward compatible with middleboxes. Multipath TCP, described in section 4, enables transmission of data segments within a transport connection over multiple network paths. Minion, described in section 5, extends TCP and SSL/TLS [35] to provide richer services to the application—unordered message delivery and multi-streaming—without changing the protocols' wire-format.

2 Providing the transport service

As explained earlier, several services have been defined in the transport layer. In this section, we first review the connectionless service. Then we present in more detail how TCP and SCTP provide a connection-oriented service and highlight the recent evolution of the key functions of these protocols. This section concludes with a discussion of the request-response service.

2.1 Providing the connectionless service

To provide a connectionless service, the transport layer mainly needs to provide some multiplexing on top of the underlying network layer. In UDP, this multiplexing is achieved by using port numbers. The 8 byte UDP header contains the source and destination port numbers that identify the applications that exchange data on the two communicating hosts. In addition to these port numbers, the UDP header contains a checksum that optionally covers the payload, the UDP header and a part of the IP header. When UDP is used above IPv4, the checksum is optional [88]. The sending application decides whether the UDP payload will be protected by a checksum. If not, the checksum field is set to zero. When UDP is used above IPv6, the checksum is mandatory and cannot be disabled by the sender.

UDP has barely changed since the publication of [88]. The only significant modification has been the UDP-lite protocol [70]. UDP-lite was designed for applications that could benefit from the delivery of possibly corrupted data. For this, UDP-lite allows the application to specify which part of the payload must be covered by a checksum. The UDP-lite header includes a checksum coverage field that indicates the part of the payload that is covered by the checksum.

2.2 Providing the connection-oriented service

The connection-oriented service is both more complex and also more frequently used. TCP and SCTP are examples of current Internet protocols that provide this service. Older protocols like TP4 or XTP [108] also provide a connection-oriented service.

The connection-oriented service can be divided in three phases :

- the establishment of the connection
- the data transfer
- the release of the connection

2.2.1 Connection establishment

The first objective of the transport layer is to multiplex connections initiated by different applications. This requires the ability to unambiguously identify different connections on the same host. TCP uses four fields that are present in the IP and TCP headers to uniquely identify a connection:

- the source IP address
- the destination IP address
- the source port
- the destination port

The source and destination addresses are the network layer addresses (e.g. IPv4 or IPv6 in the case of TCP) that have been allocated to the communicating hosts. When a connection is established by a client, the destination port is usually a well-known port number that is bound to the server application. On the other hand, the source port is often chosen randomly by the client [69]. This random selection of the source port by the client has some security implications as discussed in [5]. Since a TCP connection is identified unambiguously by using this four-tuple, a client can establish multiple connections to the same server by using different source ports on each of these connections.

The classical way of establishing a connection between two transport entities is the three-way handshake which is used by TCP [89]. This three handshake was mainly designed to deal with host crashes. It assumes that the underlying network is able to guarantee that a packet will never remain inside the network for longer than the Maximum Segment Lifetime (MSL)[1]. Furthermore, a host should not immediately reuse the same port number for subsequent connections to the same host. The TCP header contains flags that specify the role of each segment. For example, the `ACK` flag indicates that the segment contains a valid acknowledgment number while the `SYN` flag is used during the three-way handshake. To establish a connection, the original TCP specification [89] required the client to send a TCP segment with the `SYN` flag sent, including an initial sequence number extracted from a clock. According to [89], this clock had to be implemented by using a 32-bit counter incremented at least once every 4 microseconds and after each TCP connection establishment attempt. While this solution was sufficient to deal with random host crashes, it was not acceptable from a security viewpoint [48]. When a clock is used to generate the initial sequence number for each TCP connection, an attacker that wishes to inject segments inside an established connection could easily guess the sequence number to be used. To solve this problem, modern TCP implementations generate a random initial sequence number [48].

An example of the TCP three-way handshake is presented in figure 1. The client sends a segment with the `SYN` flag set (also called a `SYN` segment). The server replies with a segment that has both the `SYN` and the `ACK` flags set (a `SYN+ACK` segment). This `SYN+ACK` segment contains contains a random sequence number chosen by the server and its acknowledgment number is set to the value of the initial sequence number chosen by the client incremented by one[2]. The client replies to the `SYN+ACK` segment with an `ACK` segment that acknowledges the received `SYN+ACK` segment. This concludes the three-way handshake and the TCP connection is established.

The duration of the three-way handshake is important for applications that exchange small amount of data such as requests for small web objects. It can become longer if losses occur because TCP can only

[1] The default MSL duration is 2 minutes [89].

[2] TCP's acknowledgment number always contains the next expected sequence number and the `SYN` flag consumes one sequence number.

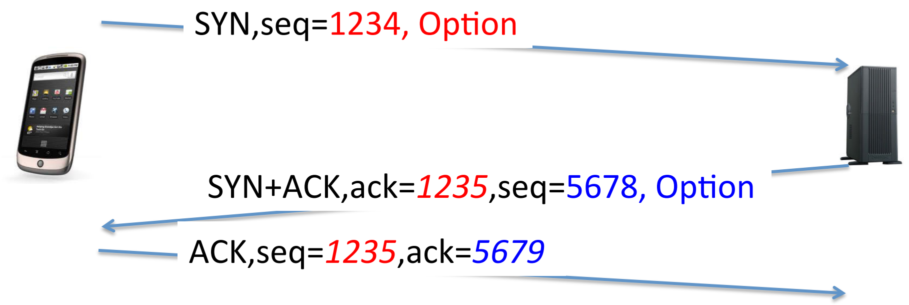

Figure 1: TCP three-way handshake

rely on its retransmission timer to recover from the loss of a SYN or SYN+ACK segment. When a client sends a SYN segment to a server, it can only rely on the initial value of its retransmission timer to recover from losses[3]. Most TCP/IP stacks have used an initial retransmission timer set to 3 seconds [18]. This conservative value was chosen in the 1980s and confirmed in the early 2000s [81]. However, this default implies that with many TCP/IP stacks, the loss of any of the first two segments of a three-way handshake will cause a delay of 3 seconds on a connection that may normally be shorter than that. Measurements conducted on large web farms showed that this initial timer had a severe impact on the performance perceived by the end users [24]. This convinced the IETF to decrease the recommended initial value for the retransmission timer to 1 second [82]. Some researchers proposed to decrease even more the value of the retransmission timer, notably in datacenter environments [111].

Another utilization of the three-way handshake is to negotiate options that are applicable for this connection. TCP was designed to be extensible. Although it does not carry a version field, in contrast with IP for example, TCP supports the utilization of options to both negotiate parameters and extend the protocol. TCP options in the SYN segment allow to negotiate the utilization of a particular TCP extension. To enable a particular extension, the client places the corresponding option inside the SYN segment. If the server replies with a similar option in the SYN+ACK segment, the extension is enabled. Otherwise, the extension is disabled on this particular connection. This is illustrated in figure 1.

Each TCP option is encoded by using a Type-Length-Value (TLV) format, which enables a receiver to silently discard the options that it does not understand. Unfortunately, there is a limit to the maximum number of TCP options that can be placed inside the TCP header. This limit comes from the Data Offset field of the TCP header that indicates the position of the first byte of the payload measured as an integer number of four bytes word starting from the beginning of the TCP header. Since this field is encoded in four bits, the TCP header cannot be longer than 60 bytes, including all options. This size was considered to be large enough by the designers of the TCP protocol, but is becoming a severe limitation to the extensibility of TCP.

A last point to note about the three-way handshake is that the first TCP implementations created state upon reception of a SYN segment. Many of these implementations also used a small queue to store the TCP connections that had received a SYN segment but not yet the third ACK. For a normal TCP connection, the

[3] If the client has sent packets earlier to the same server, it might have stored some information from the previous connection [109, 11] and use this information to bootstrap its initial timer. Recent Linux TCP/IP stacks preserve some state variables between connections.

delay between the reception of a SYN segment and the reception of the third ACK is equivalent to a round-trip-time, usually much less than a second. For this reason, most early TCP/IP implementations chose a small fixed size for this queue. Once the queue was full, these implementations dropped all incoming SYN segments. This fixed-sized queue was exploited by attackers to cause denial of service attacks. They sent a stream of spoofed SYN segment[4] to a server. Once the queue was full, the server stopped accepting SYN segments from legitimate clients [37]. To solve this problem, recent TCP/IP stacks try to avoid maintaining state upon reception of a SYN segment. This solution is often called *syn cookies*.

The principles behind *syn cookies* are simple. To accept a TCP connection without maintaining state upon reception of the SYN segment, the server must be able to check the validity of the third ACK by using only the information stored inside this ACK. A simple way to do this is to compute the initial sequence number used by the server from a hash that includes the source and destination addresses and ports and some random secret known only by the server. The low order bits of this hash are then sent as the initial sequence number of the returned SYN+ACK segment. When the third ACK comes back, the server can check the validity of the acknowledgment number by recomputing its initial sequence number by using the same hash [37]. Recent TCP/IP stacks use more complex techniques to deal notably with the options that are placed inside the SYN and need to be recovered from the information contained in the third ACK that usually does not contain any option.

At this stage, it is interesting to look at the connection establishment scheme used by the SCTP protocol [105]. SCTP was designed more than two decades after TCP and thus has benefited from several of the lessons learned from the experience with TCP. A first difference between TCP and SCTP are the segments that these protocols use. The SCTP header format is both simpler and more extensible than the TCP header.

The first four fields of the SCTP header (Source and Destination ports, Verification tag and Checksum) are present in all SCTP segments. The source and destination ports play the same role as in TCP. The verification tag is a random number chosen when the SCTP connection is created and placed in all subsequent segments. This verification tag is used to prevent some forms of packet spoofing attacks [105]. This is an improvement compared to TCP where the validation of a received segment must be performed by checking the sequence numbers, acknowledgment numbers and other fields of the header [47]. The SCTP checksum is a 32 bits CRC that provides stronger error detection properties than the Internet checksum used by TCP [107]. Each SCTP segment can contain a variable number of chunks and there is no apriori limit to the number of chunks that appear inside a segment, except that a segment should not be longer than the maximum packet length of the underlying network layer.

The SCTP connection establishment uses several of these chunks to specify the values of some parameters that are exchanged. A detailed discussion of all these chunks is outside the scope of this document and may be found in [105]. The SCTP four-way handshake uses four segments as shown in figure 2. The first segment contains the INIT chunk. To establish an SCTP connection with a server, the client first creates some local state for this connection. The most important parameter of the INIT chunk is the *Initiation tag*. This value is a random number that is used to identify the connection on the client host for its entire lifetime. This *Initiation tag* is placed as the *Verification tag* in all segments sent by the server. This is an important change compared to TCP where only the source and destination ports are used to identify a given connection. The INIT chunk may also contain the other addresses owned by the client. The server responds by sending an INIT-ACK chunk. This chunk also contains an *Initiation tag* chosen by the server and a copy of the *Initiation tag* chosen by the client. The INIT and INIT-ACK chunks also contain an initial sequence number. A key difference between TCP's three-way handshake and SCTP's four-way hand-

[4] An IP packet is said to be spoofed if it contains a source address which is different from the IP address of the sending host. Several techniques can be used by network operators to prevent such attacks [41], but measurements show that they are not always deployed [14].

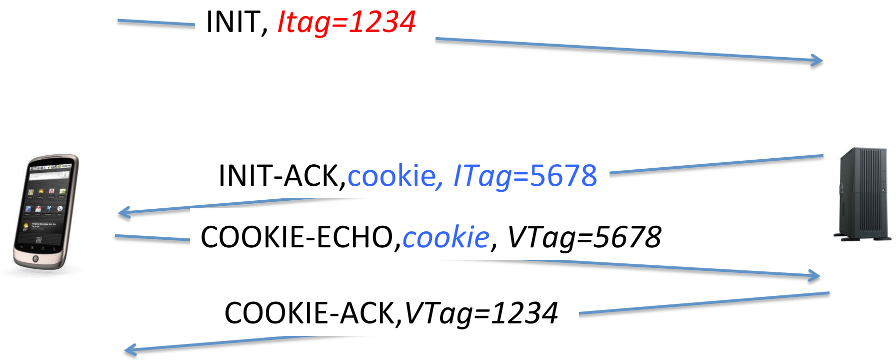

Figure 2: The four-way handshake used by SCTP

shake is that an SCTP server does not create any state when receiving an INIT chunk. For this, the server places inside the INIT-ACK reply a *State cookie* chunk. This *State cookie* is an opaque block of data that contains information from the INIT and INIT-ACK chunks that the server would have had stored locally, some lifetime information and a signature. The format of the *State cookie* is flexible and the server could in theory place almost any information inside this chunk. The only requirement is that the *State cookie* must be echoed back by the client to confirm the establishment of the connection. Upon reception of the COOKIE-ECHO chunk, the server verifies the signature of the *State cookie*. The client may provide some user data and an initial sequence number inside the COOKIE-ECHO chunk. The server then responds with a COOKIE-ACK chunk that acknowledges the COOKIE-ECHO chunk. The SCTP connection between the client and the server is now established. This four-way handshake is both more secure and more flexible than the three-way handshake used by TCP.

2.2.2 Data transfer

Before looking at the techniques that are used by transport protocols to transfer data, it is useful to look at their service models. TCP has the simplest service model. Once a TCP connection has been established, two bytestreams are available. The first bytestream allows the client to send data to the server and the second bytestream provides data transfer in the opposite direction. TCP guarantees the reliable delivery of the data during the lifetime of the TCP connection provided that it is gracefully released.

SCTP provides a slightly different service model [79]. Once an SCTP connection has been established, the communicating hosts can access two or more message streams. A message stream is a stream of variable length messages. Each message is composed of an integer number of bytes. The connection-oriented service provided by SCTP preserves the message boundaries. This implies that if an application sends a message of N bytes, the receiving application will also receive it as a single message of N bytes. This is in contrast with TCP that only supports a bytestream. Furthermore, SCTP allows the applications to use multiple streams to exchange data. The number of streams that are supported on a given connection is negotiated during connection establishment. When multiple streams have been negotiated, each application can send data over any of these streams and SCTP will deliver the data from the different streams independently without any head-of-line blocking.

While most usages of SCTP may assume an in-order delivery of the data, SCTP supports unordered delivery of messages at the receiver. Another extension to SCTP [106] supports partially-reliable delivery. With this extension, an SCTP sender can be instructed to "expire" data based on one of several events, such as a timeout, the sender can signal the SCTP receiver to move on without waiting for the "expired" data. This partially reliable service could be useful to provide timed delivery for example. With this service, there is an upper limit on the time required to deliver a message to the receiver. If the transport layer cannot deliver the data within the specified delay, the data is discarded by the sender without causing any stall in the stream.

To provide a reliable delivery of the data, transport protocols rely on various mechanisms that have been well studied and discussed in the literature : sequence numbers, acknowledgments, windows, checksums and retransmission techniques. A detailed explanation of these techniques may be found in standard textbooks [16, 99, 40]. We assume that the reader is familiar with them and discuss only some recent changes.

TCP tries to pack as much data as possible inside each segment [89]. Recent TCP stacks combine this technique with Path MTU discovery to detect the MTU to be used over a given path [73]. SCTP uses a more complex but also more flexible strategy to build its segments. It also relies on Path MTU Discovery to detect the MTU on each path. SCTP then places various chunks inside each segment. The control chunks, that are required for the correct operation of the protocol, are placed first. Data chunks are then added. SCTP can split a message in several chunks before transmission and also the bundling of different data chunks inside the same segment.

Acknowledgments allow the receiver to inform the sender of the correct reception of data. TCP initially relied exclusively on cumulative acknowledgments. Each TCP segment contains an acknowledgment number that indicates the next sequence number that is expected by the receiver. Selective acknowledgments were added later as an extension to TCP [74]. A selective acknowledgment can be sent by a receiver when there are gaps in the received data. A selective acknowledgment is simply a sequence of pairs of sequence numbers, each pair indicating the beginning and the end of a received block of data. SCTP also supports cumulative and selective acknowledgments. Selective acknowledgments are an integral part of SCTP and not an extension which is negotiated at the beginning of the connection. In SCTP, selective acknowledgments are encoded as a control chunk that may be placed inside any segment. In TCP, selective acknowledgments are encoded as TCP options. Unfortunately, given the utilization of the TCP options (notably the timestamp option [63]) and the limited space for options inside a TCP segment, a TCP segment cannot report more than three blocks of data. This adds some complexity to the handling and utilization of selective acknowledgments by TCP.

Current TCP and SCTP stacks try to detect segment losses as quickly as possible. For this, they implement various heuristics that allow to retransmit a segment once several duplicate acknowledgments have been received [40]. Selective acknowledgment also aid to improve the retransmission heuristics. If these heuristics fail, both protocols rely on a retransmission timer whose value is fixed in function of the round-trip time measured over the connection [82].

Last but not least, the transport protocols on the Internet perform congestion control. The original TCP congestion control scheme was proposed in [62]. Since then, it has evolved and various congestion control schemes have been proposed. Although the IETF recommends a single congestion control scheme [8], recent TCP stacks support different congestion control schemes and some allow the user to select the preferred one. A detailed discussion of the TCP congestion control schemes may be found in [3]. SCTP's congestion control scheme is largely similar to TCP's congestion control scheme.

Additional details about recent advances in SCTP may be found in [21]. [36] lists recent IETF documents that are relevant for TCP. [40] contains a detailed explanation of some of the recent changes to the TCP/IP protocol stack.

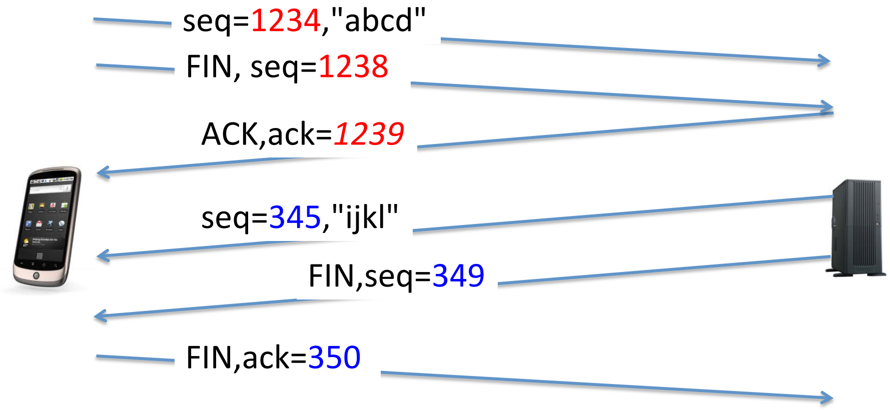

Figure 3: The four-way handshake used to close a TCP connection

2.2.3 Connection release

This phase occurs when either both hosts have exchanged all the required data or when one host needs to stop the connection for any reason (application request, lack of resources, . . .). TCP supports two mechanisms to release a connection. The main one is the four-way handshake. This handshake uses the FIN flag in the TCP header. Each host can release its own direction of data transfer. When the application wishes to gracefully close a connection, it requests the TCP entity to send a FIN segment. This segment marks the end of the data transfer in the outgoing direction and the sequence number that corresponds to the FIN flag (which consumes one sequence number) is the last one to be sent over this connection. The outgoing stream is closed as soon as the sequence number corresponding to the FIN flag is acknowledged. The remote TCP entity can use the same technique to close the other direction [89]. This graceful connection release has one advantage and one drawback. On the positive side, TCP provides a reliable delivery of all the data provided that the connection is gracefully closed. On the negative side, the utilization of the graceful release forces the TCP entity that sent the last segment on a given connection to maintain state for some time. On busy servers, a large number of connections can remain for a long time [39]. To avoid maintaining such state after a connection has been closed, web servers and some browsers send a RST segment to abruptly close TCP connections. In this case, the underlying TCP connection is closed once all the data has been transferred. This is faster, but there is no guarantee about the reliable delivery of the data.

SCTP uses a different approach to terminante connections. When an application requests a shutdown of a connection, SCTP performs a three-way handshake. This handshake uses the SHUTDOWN, SHUTDOWN-ACK and SHUTDOWN-COMPLETE chunks. The SHUTDOWN chunk is sent once all outgoing data has been acknowledged. It contains the last cumulative sequence number. Upon reception of a SHUTDOWN chunk, an SCTP entity informs its application that it cannot accept anymore data over this connection. It then ensures that all outstanding data have been delivered correctly. At that point, it sends a SHUTDOWN-ACK to confirm the reception of the SHUTDOWN segment. The three-way handshake completes with the transmission of the SHUTDOWN-COMPLETE chunk [105].

SCTP also provides the equivalent to TCP's RST segment. The ABORT chunk can be used to refuse a connection, react to the reception of an invalid segment or immediately close a connection (e.g. due to lack

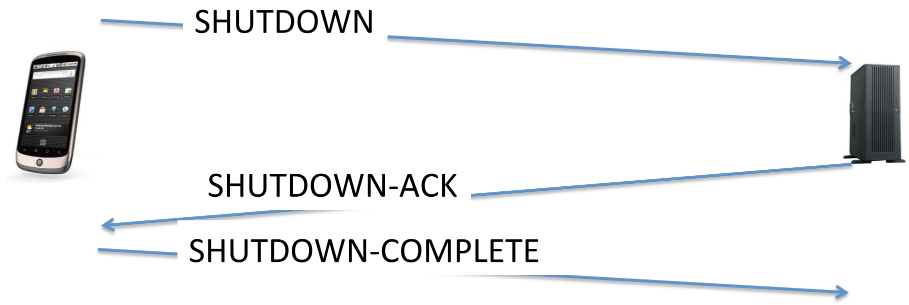

Figure 4: The three-way handshake used to close an SCTP connection

of resources).

2.3 Providing the request-response service

The request-response service has been a popular service since the 1980s. At that time, many request-response applications were built above the connectionless service, typically UDP [15]. A request-response application is very simple. The client sends a request to a server and blocks waiting for the response. The server processes the request and returns a response to the client. This paradigm is often called Remote Procedure Call (RPC) since often the client calls a procedure running on the server.

The first implementations of RPC relied almost exclusively on UDP to transmit the request and responses. In this case, the size of the requests and responses was often restricted to one MTU. In the 1980s and the beginning of the 1990s, UDP was a suitable protocol to transport RPCs because they were mainly used in Ethernet LANs. Few users were considering the utilization of RPC over the WAN. In such networks, CSMA/CD regulated the access to the LAN and there were almost no losses. Over the years, the introduction of Ethernet switches has both allowed Ethernet networks to grow in size but also implied a growing number of packet losses. Unfortunately, RPC running over UDP does not deal efficiently with packet losses because many implementation use large timeouts to recover for packet losses. TCP could deal with losses, but it was considered to be too costly for request-response applications. Before sending a request, the client must first initiate the connection. This requires a three-way handshake and thus “wastes” one round-trip-time. Then, TCP can transfer the request and receive the response over the established connection. Eventually, it performs a graceful shutdown of the connection. This connection release requires the exchange of four (small) segments, but also forces the client to remain in the `TIME_WAIT` state for a duration of 240 seconds, which limits the number of connections (and thus RPCs) that it can establish with a given server.

TCP for Transactions or T/TCP [19] was a first attempt to enable TCP to better support request/response applications. T/TCP solved the above problem by using three TCP options. These options were mainly used to allow each host to maintain an additional state variable, Connection Count (CC) that is incremented by one for every connection. This state variable is sent in the `SYN` segment and cached by the server. If a `SYN` received from a client contains a CC that is larger than the cached one, the new connection is immediately established and data can be exchanged directly (already in the `SYN`). Otherwise, a normal three-way handshake is used. The use of this state variable allowed T/TCP to reduce the duration of the `TIME_WAIT` state. T/TCP used `SYN` and `FIN` flags in the segment sent by the client and returned by the

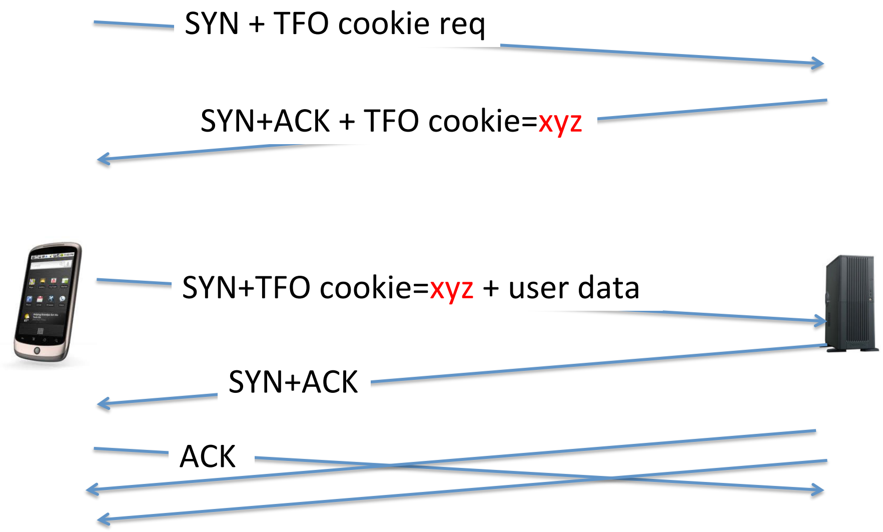

Figure 5: TCP fast open

server, which led to a two segment connection, the best solution from a delay viewpoint for RPC applications. Unfortunately, T/TCP was vulnerable to spoofing attacks [32]. An attacker could observe the Connection Count by capturing packets. Since the server only checked that the value of the CC state variable contained in a SYN segment was higher than the cached one, it was easy to inject new segments. Due to this security problem, T/TCP is now deprecated.

Improving the performance of TCP for request/response applications continued to be an objective for many researchers. However, recently the focus of the optimizations moved from the LANs that were typical for RPC applications to the global Internet. The motivation for several of the recent changes to the TCP protocol was the perceived performance of TCP with web search applications [24]. A typical web search is also a very short TCP connection during which a small HTTP request and a small HTTP response are exchanged. A first change to TCP was the increase of the initial congestion window [25]. For many years, TCP used an initial window between 2 and 4 segments [6]. This was smaller than the typical HTTP response from a web search engine [24]. Recent TCP stacks use an initial congestion window of 10 segments [25].

Another change that has been motivated by web search applications is the TCP Fast Open (TFO) extension [91]. This extension can be considered as a replacement for T/TCP. TCP fast open also enables a client to send data inside the SYN segment. TCP fast open relies on state sharing between the client and the server, but the state is more secure than the simple counter used by T/TCP. To enable the utilization of TCP fast open, the client must first obtain a *cookie* from the server. This is done by sending a SYN segment with the TFO cookie request option. The server then generates a secure cookie by encrypting the IP address of the client with a local secret [91]. The encrypted information is returned inside a TFO cookie option in the SYN+ACK segment. The client caches the cookie and associates it with the server's IP address. The subsequent connections initiated by the client will benefit from TCP fast open. The client includes the cached cookie and optional data inside its SYN segment. The server can validate the segment by decrypting its cookie. If the cookie is valid, the server acknowledges the SYN and the data that it contains. Otherwise, the optional data is ignored and a normal TCP three-way handshake is used. This is illustrated in figure 5.

3 Today's Internet

The TCP/IP protocol suite was designed with the end-to-end principle [100] in mind. TCP and SCTP are no exception to this rule. They both assume that the network contains relays that operate only at the physical, datalink and network layers of the reference models.

In such an end-to-end Internet, the payload of an IP packet is never modified inside the network and any transport protocol can be used above IPv4 or IPv6. Today, this behavior corresponds to some islands in the Internet like research backbones and some university networks. Measurements performed in enterprise, cellular and other types of commercial networks reveal that IP packets are processed differently in deployed networks [58, 115].

In addition to the classical repeaters, switches and routers, currently deployed networks contain various types of middleboxes [22]. Middleboxes were not part of the original TCP/IP architecture and they have evolved mainly during the last decade. A recent survey in enterprise networks reveals that such networks contain sometimes as many middleboxes as routers [103].

A detailed survey of all possible middleboxes is outside the scope of this chapter, but it is useful to study the operation of some important types of middleboxes to understand their impact on transport protocols and how transport protocols have to cope with them.

3.1 Firewalls

Firewalls perform checks on the received packets and decide to accept or discard them based on configured security policies. Firewalls play an important role in delimiting network boundaries and controlling incoming and outgoing traffic in enterprise networks. In theory, firewalls should not directly affect transport protocols, but in practice, they may block the deployment of new protocols or extensions to existing ones. Firewalls can either filter packets on the basis of a white list, i.e. an explicit list of allowed communication flows, or a black list, i.e. an explicit list of all forbidden communication flows. Most enterprise firewalls use a white list approach. The network administrator defines a set of allowed communication flows, based on the high-level security policies of the enterprise and configures the low-level filtering rules of the firewall to implement these policies. With such a whitelist, all flows that have not been explicitly defined are forbidden and the firewall discards all packets that do not match an accepted communication flow. This unfortunately implies that a packet that contains a different *Protocol* than the classical TCP, ICMP and UDP protocols will usually not be accepted by such a firewall. This is a major hurdle for the deployment of new transport protocols like SCTP.

Some firewalls can perform more detailed verification and maintain state for each established TCP connection. Some of these stateful firewalls are capable of verifying whether a packet that arrives for an accepted TCP connection contains a valid sequence number. For this, the firewall maintains state for each TCP connection that it accepts and when a new data packet arrives, it verifies that it belongs to an established connection and that its sequence number fits inside the advertised receive window. This verification is intended to protect the hosts that reside behind the firewall from packet injection attacks despite the fact that these hosts also need to perform the same verification.

Stateful firewalls may also limit the extensibility of protocols like TCP. To understand the problem, let us consider the large windows extension defined in [63]. This extension fixes one limitation of the original TCP specification. The TCP header [89] includes a 16-bits field that encodes the receive window in the TCP header. A consequence of this choice is that the standard TCP cannot support a receive window larger than 64 KBytes. This is not large enough for high bandwidth networks. To allow hosts to use a larger window, [63] changes the semantics of the *receive window* field of the TCP header on a per-connection basis. [63] defines the `WScale` TCP option that can only be used inside the `SYN` and `SYN+ACK` segments. This extension allows the communicating hosts to maintain their receive window as a 32 bits field. The `WScale` option contains as parameter the number of bits that will be used to shift the 32-bits window to the right before placing the lower 16 bits in the TCP header. This shift is used on a TCP connection provided that both the client and the server have included the `WScale` option in the `SYN` and `SYN+ACK` segments.

Unfortunately, a stateful firewall that does not understand the `WScale` option, may cause problems. Consider for example a client and a server that use a very large window. During the three-way handshake, they indicate with the `WScale` option that they will shift their window by 14 bits to the right. When the connection starts, each host reserves 2^{17} bytes of memory for its receive window[5]. Given the negotiated shift, each host will send in the TCP header a window field set to `0000000000000100`. If the stateful firewall does understand the `WScale` option used in the `SYN` and `SYN+ACK` segments, it will assume a window of 4 bytes and will discard all received segments. Unfortunately, there are still today stateful firewalls[6] that do not understand this TCP option defined in 1992.

Stateful firewalls can perform more detailed verification of the packets exchanged during a TCP connection. For example, intrusion detection and intrusion prevention systems are often combined with traffic normalizers [113, 53]. A traffic normalizer is a middlebox that verifies that all packets obey the protocol specification. When used upstream of an intrusion detection system, a traffic normalizer can for example buffer the packets that are received out-of-order and forward them to the IDS once they are in-sequence.

3.2 Network Address Translators

A second, and widely deployed middlebox, is the Network Address Translator (NAT). The NAT was defined in [38] and various extensions have been developed over the years. One of the initial motivation for NAT was to preserve public IP addresses and allow a group of users to share a single IP address. This enabled the deployment of many networks that use so-called *private addresses* [96] internally and rely on NATs to reach the global Internet. At some point, it was expected that the deployment of IPv6 would render NAT obsolete. This never happened and IPv6 deployment is still very slow [34]. Furthermore, some network administrators have perceived several benefits with the deployment of NATs [51] including : reducing the exposure of addresses, topology hiding, independence from providers, ...Some of these perceived advantages have caused the IETF to consider NAT for IPv6 as well, despite the available address space. Furthermore, the IETF, vendors and operators are considering the deployment of large scale Carrier Grade NATs (CGN) to continue to use IPv4 despite the depletion of the addressing space [83] and also to ease the deployment of IPv6 [64]. NATs will remain a key element of the deployed networks for the foreseeable future.

There are different types of NATs depending on the number of addresses that they support and how they maintain state. In this section, we concentrate on a simple but widely deployed NAT that serves a large number of users on a LAN and uses a single public IP address. In this case, the NAT needs to map several private IP addresses on a single public address. To perform this translation and still allow several internal hosts to communicate simultaneously, the NAT must understand the transport protocol that is used by the internal hosts. For TCP, the NAT needs to maintain a pool of TCP port numbers and use one of the available port as the source port for each new connection initiated by an internal host. Upon reception of a packet, the NAT needs to update the source and destination addresses, the source (or destination) port number and

[5] It is common to start a TCP connection with a small receive window/buffer and automatically increase the buffer size during the transfer [102].

[6] See e.g. http://support.microsoft.com/kb/934430

also the IP and TCP checksums. The NAT performs this modification transparently in both directions. It is important to note that a NAT can only change the header of the transport protocol that it supports. Most deployed NATs only support TCP [49], UDP [9] and ICMP [104]. Supporting another transport protocol on a NAT requires software changes [55] and few NAT vendors implement those changes. This often forces users of new transport protocol to tunnel their protocol on top of UDP to traverse NATs and other middleboxes [85, 110]. This limits the ability to innovate in the transport layer.

NAT can be used transparently by most Internet applications. Unfortunately, some applications cannot easily be used over NATs [57]. The textbook example of this problem is the File Transfer Protocol (FTP) [90]. An FTP client uses two types of TCP connections : a control connection and data connections. The control connection is used to send commands to the server. One of these is the `PORT` command that allows to specify the IP address and the port numbers that will be used for the data connection to transfer a file. The parameters of the `PORT` command are sent using a special ASCII syntax [90]. To preserve the operation of the FTP protocol, a NAT can translate the IP addresses and ports that appear in the IP and TCP headers, but also as parameters of the `PORT` command exchanged over the data connection. Many deployed NATs include Application Level Gateways (ALG) [57] that implement part of the application level protocol and modify the payload of segments. For FTP, after translation a `PORT` command may be longer or shorter than the original one. This implies that the FTP ALG needs to maintain state and will have to modify the sequence/acknowledgment numbers of all segments sent over a connection after having translated a `PORT` command. This is transparent for the FTP application, but has influenced the design of Multipath TCP although recent FTP implementations rarely use the `PORT` command[7].

3.3 Proxies

The last middlebox that we cover is the proxy. A proxy is a middlebox that resides on a path and terminates TCP connections. A proxy can be explicit or transparent. The SOCKS5 protocol [71] is an example of the utilization of an explicit proxy. SOCKS5 proxies are often used in enterprise networks to authenticate the establishment of TCP connections. A classical example of transparent proxies are the HTTP proxies that are deployed in various commercial networks to cache and speedup HTTP requests. In this case, some routers in the network are configured to intercept the TCP connections and redirect them to a proxy server [75]. This redirection is transparent for the application, but from a transport viewpoint, the proxy acts as a relay between the two communicating hosts and there are two different TCP connections[7]. The first one is initiated by the client and terminates at the proxy. The second one is initiated by the proxy and terminates at the server. The data exchanged over the first connection is passed to the second one, but the TCP options are not necessarily preserved. In some deployments, the proxy can use different options than the client and/or the server.

3.4 How prevalent are middleboxes?

We've learnt that middleboxes of various kinds exist, but are they really deployed in today's networks? Answering this question can guide the right way to develop new protocols and enhance existing ones.

Here we briefly review measurement studies that have attempted to paint an accurate image of middlebox deployments in use. The conclusion is that middleboxes are widespread to the point where end-to-end paths without middleboxes have become exceptions, rather than the norm.

[7] Some deployments use several proxies in cascade. This allows the utilization of compression techniques and other non-standard TCP extensions on the connection between two proxies.

There are two broad types of middlebox studies. The first type of study uses ground-truth topology information from providers and other organizations. While accurate, these studies may overestimate the influence of middleboxes on end-to-end traffic because certain behavior is only triggered in rare, corner cases (e.g. an intrusion prevention system may only affect traffic carrying known worm signatures). These studies do not tell us exactly *what operations* are applied to packets by middleboxes. One recent survey study has found that the 57 enterprise networks surveyed deploy as many middleboxes as routers [103].

Active measurements probe end-to-end paths to trigger "known" behaviors of middleboxes. Such measurements are very accurate, pinpointing exactly what middleboxes do in certain scenarios. However, they offer a lower bound of middlebox deployments, as they may pass through other middleboxes without triggering them. Additionally, such studies are limited to probing a full path (cannot probe path segments) thus cannot tell how many middleboxes are deployed on a path exhibiting middlebox behavior. The data surveyed below comes from such studies.

Network Address Translators are easy to test for: the traffic source needs to compare its local source address with the source address of its packets reaching an external site (e.g. what's my IP). When the addresses differ, a NAT has been deployed on path. Using this basic technique, existing studies have shown that NATs are deployed almost universally by (or for) end-users, be they home or mobile:

- Most cellular providers use them to cope with address space shortage and to provide some level of security. A study surveying 107 cellular networks across the globe found that 82 of them used NATs [115].

- Home users receive a single public IP address (perhaps via DHCP) from their access providers, and deploy NATs to support multiple devices. The typical middlebox here is the "wireless router", an access point that aggregates all home traffic onto the access link. A study using peer-to-peer clients found that only 12% of the peers have a public IP address [30].

Testing for stateless firewalls is equally simple: the client generates traffic using different transport protocols and port numbers, and the server acks it back. As long as some tests work, the client knows the endpoint is reachable, and that the failed tests are most likely due to firewall behavior. There is a chance that the failed tests are due to the stochastic packet loss inherent in the Internet; that is why tests are interleaved and run multiple times.

Firewalls are equally widespread in the Internet, being deployed by most cellular operators [115]. Home routers often act as firewalls, blocking new protocols and even existing ones (e.g. UDP) [30]. Most servers also deploy firewalls to restrict in-bound traffic [103]. Additionally, many modern operating systems come with "default-on" firewalls. For instance, the Windows 7 firewall explicitly asks users to allow incoming connections and to whitelist applications allowed to make outgoing connections.

Testing for explicit proxies can be done by comparing the segments that leave the source with the ones arriving at the destination. A proxy will change sequence numbers, perhaps segment packets differently, modify the receive window, and so forth. Volunteers from various parts of the globe ran TCPExposure, a tool that aims to detect such proxies and other middlebox behavior [58]. The tests cover 142 access networks (including cellular, DSL, public hotspots and offices) in 24 countries. Proxying behavior was seen on 10% of the tested paths[58].

Beyond basic reachability, middleboxes such as traffic normalizers and stateful firewalls have some expectations on the packets they see: what exactly do these middleboxes do, and how widespread are they? The same study sheds some light into this matter:

- Middlebox behavior depends on the ports used. Most middleboxes are active on port 80 (HTTP traffic).

- A third of paths keep TCP flow state and use it to actively correct acknowledgments for data the middlebox has not seen. To probe this behavior, TCPExposure sends a few TCP segments leaving a gap in the sequence number, while the server acknowledgment also covers the gap.

- 14% of paths remove unknown options from SYN packets. These paths will not allow TCP extensions to be deployed.

- 18% of paths modify sequence numbers; of these, a third seem to be proxies, and the other are most likely firewalls than randomize the initial sequence number to protect vulnerable end hosts against in-window injection attacks.

3.5 The Internet is Ossified

Deploying a new IP protocol requires a lot of investment to change all the deployed hardware. Experience with IPv6 after 15 years since it has been standardized is not encouraging: only a minute fraction of the Internet has migrated to v6, and there are no signs of it becoming "the Internet" anytime soon.

Changing IPv4 itself is in theory possible with IP options. Unfortunately, it has been known for a while now that "IP options are not an option" [42]. This is because existing routers implement forwarding in hardware for efficiency reasons; packets carrying unknown IP options are treated as exceptions that are processed in software. To avoid a denial-of-service on routers' CPU's, such packets are dropped by most routers.

In a nutshell, we can't really touch IP - but have we also lost our ability to change transport protocols as well? The high level picture emerging from existing middlebox studies is that of a network that is highly tailored to today's traffic to the point it is ossified: *changing existing transport protocols is challenging as it needs to carefully consider middlebox interactions*. Further, *deploying new transport protocols natively is almost impossible*.

Luckily, changing transport protocols is still possible, albeit great care must be taken when doing so. Firewalls will block *any* traffic they do not understand, so deploying new protocols must necessarily use existing ones just to get through the network. This observation has recently lead to the development of Minion, a container protocol that enables basic connectivity above TCP while avoiding its in-order, reliable bytestream semantics at the expense of slightly increased bandwidth usage. We discuss Minion in Section 5.

Even changing TCP is very difficult. The semantics of TCP are embedded in the network fabric, and new extensions must function within the confines of these semantics, or they will fail. In section 4 we discuss Multipath TCP, another recent extension to TCP, that was designed explicitly to be middlebox compatible.

4 Multipath TCP

Today's networks are multipath: mobile devices have multiple wireless interfaces, datacenters have many redundant paths between servers and multi-homing has become the norm for big server farms. Meanwhile, TCP is essentially a single path protocol: when a TCP connection is established, it is bound to the IP addresses of the two communicating hosts. If one of these addresses changes, for whatever reason, the connection fails. In fact a TCP connection cannot even be load-balanced across more than one path within the network, because this results in packet reordering and TCP misinterprets this reordering as congestion and slows down.

This mismatch between today's multipath networks and TCP's single-path design creates tangible problems. For instance, if a smartphone's WiFi interface loses signal, the TCP connections associated with it stall - there is no way to migrate them to other working interfaces, such as 3G. This makes mobility a frustrating experience for users. Modern datacenters are another example: many paths are available between two endpoints, and equal cost multipath routing randomly picks one for a particular TCP connection. This can cause collisions where multiple flows get placed on the same link, hurting throughput - to such an extent that average throughput is halved in some scenarios.

Multipath TCP (MPTCP) [13] is a major modification to TCP that allows multiple paths to be used simultaneously by a single connection. Multipath TCP circumvents the issues above and several others that affect TCP. Changing TCP to use multiple paths is not a new idea: it was originally proposed more than fifteen years ago by Christian Huitema in the Internet Engineering Task Force (IETF) [59], and there have been a half-dozen more proposals since then to similar effect. Multipath TCP draws on the experience gathered in previous work, and goes further to solve issues of fairness when competing with regular TCP and deployment issues due to middleboxes in today's Internet.

4.1 Overview of Multipath TCP

The design of Multipath TCP has been influenced by many requirements, but there are two that stand out: application compatibility and network compatibility. Application compatibility implies that applications that today run over TCP should work without any change over Multipath TCP. Next, Multipath TCP must operate over any Internet path where the TCP protocol operates.

As explained earlier, many paths on today's Internet include middleboxes that, unlike routers, know about the TCP connections they forward, and affect them in special ways. Designing TCP extensions that can safely traverse all these middleboxes has proven to be challenging.

Multipath TCP allows multiple ”subflows” to be combined to form a single MPTCP session. An MPTCP session starts with an initial subflow which is very similar to a regular TCP connection, with a three way handshake. After the first MPTCP subflow is set up, additional subflows can be established. Each subflow also looks very similar to a regular TCP connection, complete with three-way handshake and graceful teardown, but rather than being a separate connection it is bound into an existing MPTCP session. Data for the connection can then be sent over any of the active subflows that has the capacity to take it.

To examine Multipath TCP in more detail, let us consider a very simple scenario with a smartphone client and a single-homed server. The smartphone has two network interfaces: a WiFi interface and a 3G interface; each has its own IP address. The server, being single-homed, has a single IP address. In this environment, Multipath TCP would allow an application on the mobile device to use a single MPTCP session that can use both the WiFi and the 3G interfaces to communicate with the server. The application opens a regular TCP socket, and the kernel enables MPTCP by default if the remote end supports it, using both paths. The application does not need to concern itself with which radio interface is working best at any instant; MPTCP handles that for it. In fact, Multipath TCP can work when both endpoints are multihomed (in this case subflows are opened between all pairs of ”compatible” IP addresses), or even in the case when both endpoints are single homed (in this case different subflows will use different port numbers, and can be routed differently by multipath routing in the network).

Connection Setup. Let us walk through the establishment of an MPTCP connection. Assume that the mobile device chooses its 3G interface to open the connection. It first sends a `SYN` segment to the server. This segment contains the `MP_CAPABLE` TCP option indicating that the mobile device supports Multipath TCP. This option also contains a key which is chosen by the mobile device. The server replies with a `SYN+ACK` segment containing the `MP_CAPABLE` option and the key chosen by the server. The mobile device completes the handshake by sending an `ACK` segment.

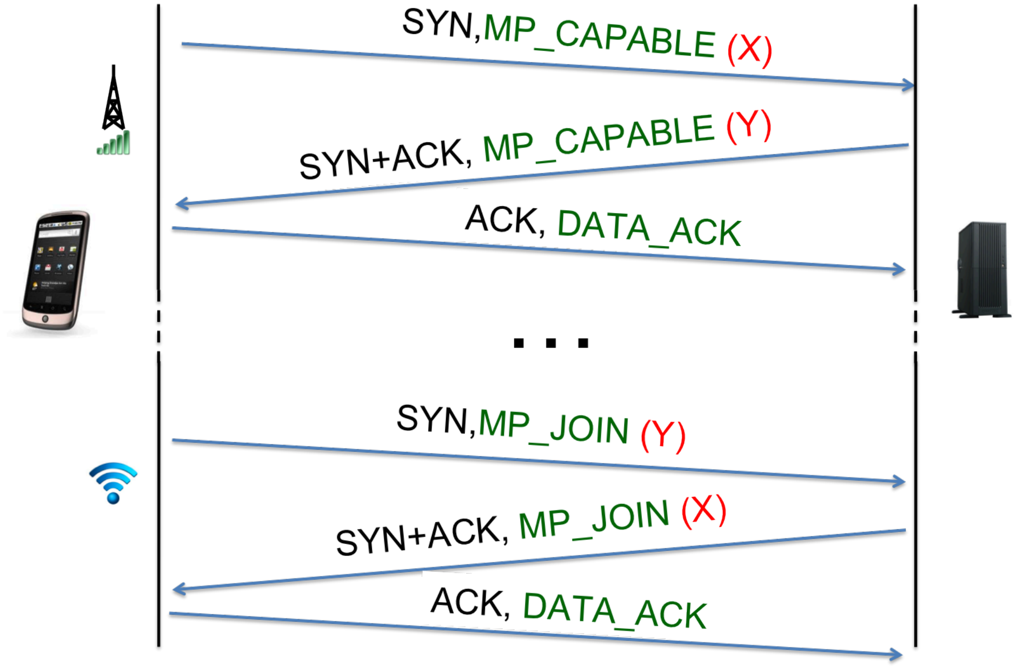

Figure 6: Multipath TCP handshake: multiple subflows can be added and removed after the initial connection is setup and connection identifiers are exchanged.

The initial MPTCP connection setup is shown graphically in the top part of Figure 6, where the segments are regular TCP segments carrying new multipath TCP-related options (shown in green).

At this point the Multipath TCP connection is established and the client and server can exchange TCP segments via the 3G path. How could the mobile device also send data through this Multipath TCP session over its WiFi interface?

Naively, it could simply send some of the segments over the WiFi interface. However most ISPs will drop these packets, as they would have the source address of the 3G interface. Perhaps the client could tell the server the IP address of the WiFi interface, and use that when it sends over WiFi? Unfortunately this will rarely work: firewalls and similar stateful middleboxes on the WiFi path expect to see a `SYN` segment before they see data segment. The only solution that will work reliably is to perform a regular three-way handshake on the WiFi path before sending any packets that way, so this is what Multipath TCP does. This handshake carries the `MP_JOIN` TCP option, providing information to the server that can securely identify the correct connection to associate this additional subflow with. The server replies with `MP_JOIN` in the `SYN+ACK`, and the new subflow is established (this is shown in the bottom part of Figure 6).

An important point about Multipath TCP, especially in the context of mobile devices, is that the set of subflows that are associated to a Multipath TCP connection is not fixed. Subflows can be dynamically added and removed from a Multipath TCP connection throughout its lifetime, without affecting the bytestream transported on behalf of the application. If the mobile device moves to another WiFi network, it will receive a new IP address. At that time, it will open a new subflow using its newly allocated address and tell the server that its old address is not usable anymore. The server will now send data towards the new address.

These options allow mobile devices to easily move through different wireless connections without breaking their Multipath TCP connections [80].

Multipath TCP also implements mechanisms that allows to inform the remote host of the addition/removal of addresses even when an endpoint operates behind a NAT, or when a subflow using a different address family is needed (e.g. IPv6). Endpoints can send an ADD_ADDR option that contains an address identifier together with an address. The address identifier is unique at the sender, and allows it to identify its addresses even when it is behind a NAT. Upon receiving an advertisement, the endpoint may initiate a new subflow to the new address. An address withdrawal mechanism is also provided via the REMOVE_ADDR option that also carries an address identifier.

Data Transfer. Assume now that two subflows have been established over WiFi and 3G: the mobile device can send and receive data segments over both. Just like TCP, Multipath TCP provides a bytestream service to the application. In fact, standard applications can function over MPTCP without being aware of it - MPTCP provides the same socket interface as TCP.

Since the two paths will often have different delay characteristics, the data segments sent over the two subflows will not be received in order. Regular TCP uses the sequence number in the TCP header to put data back into the original order. A simple solution for Multipath TCP would be to just reuse this sequence number as is.

Unfortunately, this simple solution would create problems with some existing middleboxes such as firewalls. On each path, a middlebox would only see half of the packets, so it would observe many gaps in the TCP sequence space. Measurements indicate that some middleboxes react in strange ways when faced with gaps in TCP sequence numbers [58]. Some discard the out-of-sequence segments while others try to update the TCP acknowledgments in order to ”recover” some of these gaps. With such middleboxes on a path, Multipath TCP cannot safely send TCP segments with gaps in the TCP sequence number space. On the other hand, Multipath TCP also cannot send every data segment over all subflows: that would be a waste of resources.

To deal with this problem, Multipath TCP uses its own sequence numbering space. Each segment sent by Multipath TCP contains two sequence numbers: the subflow sequence number inside the regular TCP header and an additional Data Sequence Number (DSN).

This solution ensures that the segments sent on any given subflow have consecutive sequence numbers and do not upset middleboxes. Multipath TCP can then send some data sequence numbers on one path and the remainder on the other path; the DSN will be used by the Multipath TCP receiver to reorder the bytestream before it is given to the receiving application.

Before we explain the way the Data Sequence Number is encoded, we first need to discuss two other key parts of Multipath TCP that are affected by the additional sequence number space—flow control and acknowledgements.

Flow Control. TCP’s receive window indicates the number of bytes beyond the sequence number from the acknowledgment field that the receiver can buffer. The sender is not permitted to send more than this amount of additional data. Multipath TCP also needs to implement flow control, although segments can arrive over multiple subflows. If we inherit TCP’s interpretation of receive window, this would imply an MPTCP receiver maintains a pool of buffering per subflow, with receive window indicating per-subflow buffer occupancy. Unfortunately such an interpretation can lead to deadlocks:

1. The next segment that needs to be passed to the application was sent on subflow 1, but was lost.

2. In the meantime subflow 2 continues delivering data, and fills its receive window.

3. Subflow 1 fails silently.
4. The missing data needs to be re-sent on subflow 2, but there is no space left in the receive window, resulting in a deadlock.

The correct solution is to generalize TCP's receive window semantics to MPTCP. For each connection *a single receive buffer pool should be shared between all subflows*. The receive window then indicates the maximum data sequence number that can be sent rather than the maximum subflow sequence number. As a segment resent on a different subflow always occupies the same data sequence space, deadlocks cannot occur.

The problem for an MPTCP sender is that to calculate the highest data sequence number that can be sent, the receive window needs to be added to the highest data sequence number acknowledged. However the ACK field in the TCP header of an MPTCP subflow must, by necessity, indicate only subflow sequence numbers to cope with middleboxes. Does MPTCP need to add an extra data acknowledgment field for the receive window to be interpreted correctly?

Acknowledgments. The answer is positive: MPTCP needs and uses *explicit connection-level acknowledgments* or *DATA_ACKs*. The alternative is to infer connection-level acknowledgments from subflow acknowledgments, by using a scoreboard maintained by the sender that maps subflow sequence numbers to data sequence numbers. Unfortunately, MPTCP segments and their associated ACKs will be reordered as they travel on different paths, making it impossible to correctly infer the connection-level acknowledgments from subflow-level ones [94].

Encoding. We have seen that in the forward path we need to encode a mapping of subflow bytes into the data sequence space, and in the reverse path we need to encode cumulative data acknowledgments. There are two viable choices for encoding this additional data:

- Send the additional data in TCP options.
- Carry the additional data within the TCP payload, using a chunked or escaped encoding to separate control data from payload data.

For the forward path there aren't compelling arguments either way, but the reverse path is a different matter. Consider a hypothetical encoding that divides the payload into chunks where each chunk has a TLV (type-length-value) header. A data acknowledgment can then be embedded into the payload using its own chunk type. Under most circumstances this works fine. However, unlike TCP's pure ACK, anything embedded in the payload must be treated as data. In particular:

- It must be subject to flow control because the receiver must buffer data to decode the TLV encoding.
- If lost, it must be retransmitted consistently, so that middleboxes can track sequence state correctly[8]
- If packets before it are lost, it might be necessary to wait for retransmissions before the data can be parsed - causing head-of-line blocking.

Flow control presents the most obvious problem for the chunked payload encoding. Figure 7 provides an example. Client C is pipelining requests to server S; meanwhile S's application is busy sending the large

[8] TCP proxies re-send the original content they see a "retransmission" with different data.

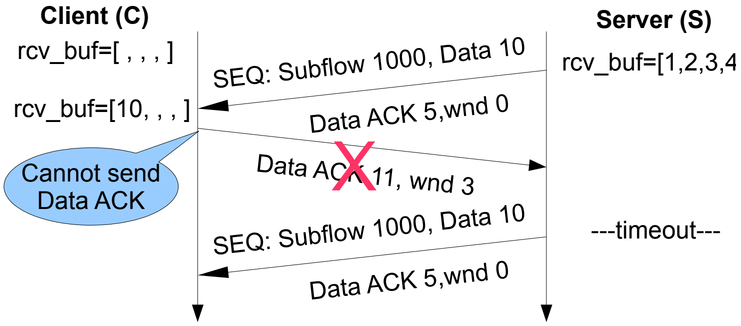

Figure 7: Flow Control on the path from C to S inadvertently stops the data flow from S to C

response to the first request so it isn't yet ready to read the subsequent requests. At this point, S's receive buffer fills up.

S sends segment 10, C receives it and wants to send the DATA_ACK, but cannot: flow control imposed by S's receive window stops him. Because no DATA_ACKs are received from C, S cannot free his send buffer, so this fills up and blocks the sending application on S. S's application will only read when it has finished sending data to C, but it cannot do so because its send buffer is full. The send buffer can only empty when S receives the DATA_ACK from C, but C cannot send this until S's application reads. This is a classic deadlock cycle. As no DATA_ACK is received, S will eventually time out the data it sent to C and will retransmit it; after many retransmits the whole connection will time out.

The conclusion is that DATA_ACKs cannot be safely encoded in the payload. The only real alternative is to encode them in TCP options which (on a pure ACK packet) are not subject to flow control.

The Data Sequence Mapping. If MPTCP must use options to encode DATA_ACKs, it is simplest to also encode the mapping from subflow sequence numbers to data sequence numbers in a TCP option. This is the *data sequence mapping* or DSM.

At first glance it seems the DSM option simply needs to carry the data sequence number corresponding to the start of the MPTCP segment. Unfortunately middleboxes and interfaces that implement TSO or LRO make this far from simple.

Middleboxes that re-segment data would cause a problem. TCP Segmentation Offload (TSO) hardware in the network interface card (NIC) also re-segments data and is commonly used to improve performance. The basic idea is that the OS sends large segments and the NIC re-segments them to match the receiver's MSS. What does TSO do with TCP options? A test of 12 NICs supporting TSO from four different vendors showed that all of them copy a TCP option sent by the OS on a large segment into all the split segments [94].

If MPTCP's DSM option only listed the data sequence number, TSO would copy the same DSM to more than one segment, breaking the mapping. Instead the DSM option must say precisely which subflow bytes map to which data sequence numbers. But this is further complicated by middleboxes that modify the initial sequence number of TCP connections and consequently rewrite all sequence numbers (many firewalls behave like this). Instead, the DSM option must map the offset from the subflow's initial sequence number

```
                     1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------+---------------+-------+----------------------+
|     Kind      |    Length     |Subtype| (reserved) |F|m|M|a|A|
+---------------+---------------+-------+----------------------+
|           Data ACK (4 or 8 octets, depending on flags)       |
+--------------------------------------------------------------+
|   Data Sequence Number (4 or 8 octets, depending on flags)   |
+--------------------------------------------------------------+
|              Subflow Sequence Number (4 octets)              |
+-------------------------------+------------------------------+
|  Data-level Length (2 octets) |      Checksum (2 octets)     |
+-------------------------------+------------------------------+
```

Figure 8: The Data Sequence Signal option in MPTCP that carries the Data Sequence Mapping information, the Data ACK, the Data FIN and connection-fall-back options.

to the data sequence number, as the offset is unaffected by sequence number rewriting. The option must also contain the length of the mapping. This is robust - as long as the option is received, it does not greatly matter which packet carries it, so duplicate mappings caused by TSO are not a problem.

Dealing with Content-Modifying Middleboxes. Multipath TCP and content-modifying middleboxes (such as application-level NATs, e.g. for FTP) have the potential to interact badly. In particular, due to FTP's ASCII encoding, re-writing an IP address in the payload can necessitate changing the length of the payload. Subsequent sequence and ACK numbers are then fixed up by the middlebox so they are consistent from the point of view of the end systems.

Such length changes break the DSM option mapping - subflow bytes can be mapped to the wrong place in the data stream. They also break every other possible mapping mechanism, including chunked payloads. There is no easy way to handle such middleboxes.

That is why MPTCP includes an optional checksum in the DSM mapping to detect such content changes. If an MPTCP host receives a segment with an invalid DSM checksum, it rejects the segment and triggers a fallback process: if any other subflows exists, MPTCP terminates the subflow on which the modification occurred; if no other subflow exists, MPTCP drops back to regular TCP behavior for the remainder of the connection, allowing the middlebox to perform rewriting as it wishes. This fallback mechanism preserves connectivity in the presence of middleboxes.

For efficiency reasons, MPTCP uses the same 16-bit ones complement checksum used in the TCP header. This allows the checksum over the payload to be calculated only once. The payload checksum is added to a checksum of an MPTCP pseudo header covering the DSM mapping values and then inserted into the DSM option. The same payload checksum is added to the checksum of the TCP pseudo-header and then used in the TCP checksum field. MPTCP allows checksums to be disabled for high performance environments such as data-centers where there is no chance of encountering such an application-level gateway.

The fall-back-to-TCP process, triggered by a checksum failure, can also be triggered in other circumstances. For example, if a routing change moves an MPTCP subflow to a path where a middlebox removes DSM options, this also triggers the fall-back procedure.

Connection Release. Multipath TCP must allow a connection to survive even though its subflows are coming and going. Subflows in MPTCP can be torn down by means of a four-way handshake as regular TCP flows—this ensures MPTCP allows middleboxes to clear their state when a subflow is not used anymore.

MPTCP uses an explicit four-way handshake for connection tear-down indicated by a `DATA_FIN` option. The `DATA_FIN` is MPTCP's equivalent to TCP's `FIN`, and it occupies one byte in the data-sequence space. A `DATA_ACK` will be used to acknowledge the receipt of the `DATA_FIN`. MPTCP requires that the segment(s) carrying a `DATA_FIN` must also have the `FIN` flag set - this ensures all subflows are also closed when the MPTCP connection is being closed.

For reference, we show the wire format of the option used by MPTCP for data exchange in Figure 8. This option encodes the Data Sequence Mapping, Data ACK, Data FIN and the fall-back options. The flags specify which parts of the option are valid, and help reduce option space usage.

4.2 Congestion Control

One of the most important components in TCP is its congestion controller which enables it to adapt its throughput dynamically in response to changing network conditions. To perform this functionality, each TCP sender maintains a congestion window w which governs the amount of packets that the sender can send without waiting for an acknowledgment. The congestion window is updated dynamically according to the following rules:

- On each ACK, increase the window w by $1/w$.
- Each loss decrease the window w by $w/2$.

TCP congestion control ensures fairness: when multiple connections utilize the same congested link each of them will independently converge to the same average value of the congestion window.

What is the equivalent of TCP congestion control for multipath transport? The obvious question to ask is why not just run regular TCP congestion control on each subflow? Consider the scenario in Fig. 9. If multipath TCP ran regular TCP congestion control on both paths, then the multipath flow would obtain twice as much throughput as the single path flow (assuming all RTTs are equal). This is unfair. To solve this problem, one solution is to try and detect shared bottlenecks but that is unreliable; a better solution is be less aggressive on each subflow (i.e. increase window slower) such that in aggregate the MPTCP connection is no more aggressive than a single regular TCP connection.

Before we describe solutions to MPTCP congestion control, let's discuss the three goals that multipath congestion control must obey [117]:

Fairness If several subflows of the same MPTCP connection share a bottleneck link with other TCP connections, MPTCP should not get more throughput than TCP.

Deployability The performance of all the Multipath TCP subflows together should be at least that of regular TCP on any of the paths used by a Multipath TCP connection. This ensures that there is an incentive to deploy Multipath TCP.

Efficiency A final, most important goal is that Multipath TCP should prefer efficient paths, which means it should send more of its traffic on paths experiencing less congestion.

Intuitively, this last goal ensures wide-area load balancing of traffic: when a multipath connection is using two paths loaded unevenly (such as Figure 10), the multipath transport will prefer the unloaded path

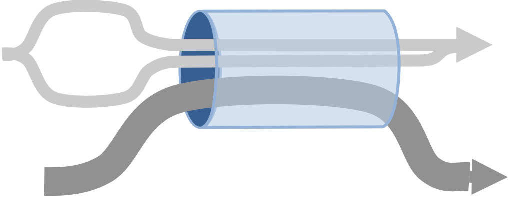

Figure 9: A scenario which shows the importance of weighting the aggressiveness of subflows (Reprinted from [117]. Included here by permission).

and push most of its traffic there; this will decrease the load on the congested link and increase it on the less congested one.

If a large enough fraction of flows are multipath, congestion will spread out evenly across collections of links, creating ”resource pools”: links that act together as if they are a single, larger capacity link shared by all flows. This effect is called resource pooling [116]. Resource pooling brings two major benefits, discussed in the paragraphs below.

Increased Fairness. Consider the example shown in Figure 10: congestion balancing ensures that all flows have **the same throughput**, making the two links of 20 pkt/s act like a single pooled link with capacity 40 pkt/s shared fairly by the four flows. If more MPTCP flows would be added, the two links would still behave as a pool, sharing capacity fairly among all flows. Conversely, if we remove the Multipath TCP flow, the links no longer form a pool, and the throughput allocation is unfair as the TCP connection using the top path gets twice as much throughput as the TCP connections using the bottom path.

Increased Throughput. Consider the somewhat contrived scenario in Fig.11, and suppose that the three links each have capacity 12Mb/s. If each flow split its traffic evenly across its two paths subflow would get 4Mb/s hence each flow would get 8Mb/s. But if each flow used only the one-hop shortest path, it could get 12Mb/s: this is because two-hop paths consume double the resources of one-hop paths, and in a congested network it makes sense to only use the one-hop paths.

In an idle network, however, using all available paths is much better: consider the case when only the blue connection is using the links. In this case this connection would get 24Mb/s throughput; using the one hop path alone would only provide 12Mb/s.

In summary, the endpoints need to be able to dynamically decide which paths to use based on conditions in the network. A solution has been devised in the theoretical literature on congestion control, independently by Kelly and Voice [65] and Han et al. [52]. The core idea is that a multipath flow should shift all its traffic onto the least-congested path. In a situation like Fig. 11 the two-hop paths will have higher drop probability than the one-hop paths, so applying the core idea will yield the efficient allocation. Surprisingly it turns out that this can be achieved by doing independent congestion control at endpoints.

Multipath TCP Congestion Control. The theoretical work on multipath congestion control [65, 52] assumes a rate-based protocol, with exponential increases of the rate. TCP, in contrast, is a packet-based protocol, sending w packets every round-trip time (i.e. the rate is w/RTT); a new packet is sent only when an acknowledgment is received, confirming that an existing packet has left the network. This property is called ACK-clocking and is nice because it has good stability properties: when congestion occurs round-trip times increase (due to buffering), which *automatically reduces the effective rate* [62].

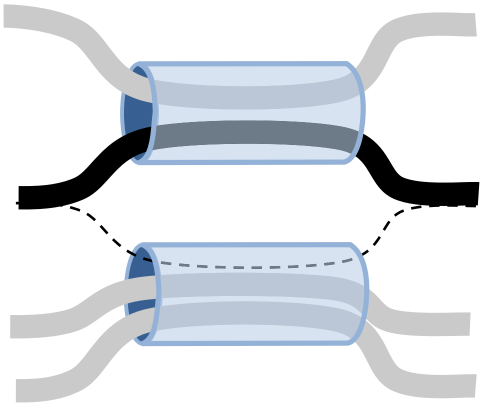

Figure 10: Two links each with capacity 20 pkts/s. The top link is used by a single TCP connection, and the bottom link is used by two TCP connections. A Multipath TCP connection uses both links. Multipath TCP pushes most of its traffic onto less congested top link, making the two links behave like a resource pool of capacity 40 pkts/s. Capacity is divided equally, with each flow having throughput 10 pkts/s. (Reprinted from [117]. Included here by permission)

Converting a theoretical rate-based exponential protocol to a practical packet-based protocol fair to TCP turned out to be more difficult than expected. There are two problems that appear [117]:

- When loss rates are equal on all paths, the theoretical algorithm will place all of the window on one path or the other, not on both—this effect was termed “flappiness” and it appears because of the discrete (stochastic) nature of packet losses which are not captured by the differential equations used in theory.
- The ideal algorithm always prefers paths with lower loss rate, but in practice these may have poor performance. Consider a mobile phone with WiFi and 3G links: 3G links have very low loss rates and huge round-trip times, resulting in poor throughput. WiFi is lossy, has shorter round-trip times and

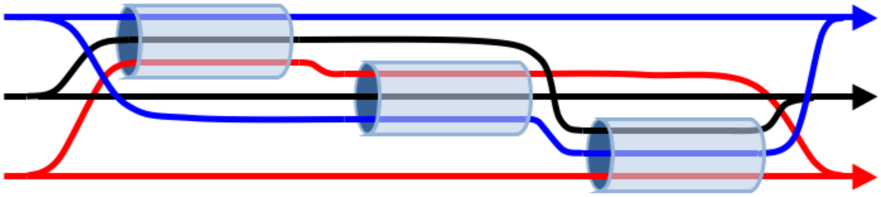

Figure 11: A scenario to illustrate the importance of choosing the less-congested path. (Reprinted from [117]. Included here by permission.)

typically offers much better throughput. In this common case, a "perfect" controller would place all traffic on the 3G path, violating the second goal (deployability).

The pragmatic choice is to sacrifice some load-balancing ability to ensure greater stability and to offer incentives for deployment. This is what Multipath TCP congestion control does.

Multipath TCP congestion control is a series of simple changes to the standard TCP congestion control mechanism. Each subflow has its own congestion window, that is halved when packets are lost, as in standard TCP [117].

Congestion balancing is implemented in the increase phase of congestion control: here Multipath TCP will allow less congested subflows to increase proportionally more than congested ones. Finally, the total increase of Multipath TCP across all of its subflows is dynamically chosen in such a way that it achieves the first and second goals above.

The exact algorithm is described below and it satisfies the goals we've discussed:

- Upon ACK on subflow r, increase the window w_r by $\min(a/w_{total}), 1/w_r)$.
- Upon loss on subflow r, decrease the window w_r by $w_r/2$.

Here

$$a = w_{total} \frac{\max_r w_r/RTT_r^2}{(\sum_r w_r/RTT_r)^2}, \tag{1}$$

w_r is the current window size on subflow r and w_{total} is the sum of windows across all subflows.

The algorithm biases the increase towards uncongested paths: these will receive more ACKs and will increase accordingly. However, MPTCP does keep some traffic even on the highly congested paths; this ensures stability and allows it to quickly detect when path conditions improve.

a is a term that is computed dynamically upon each packet drop. Its purpose is to make sure that MPTCP gets at least as much throughput as TCP on the best path. To achieve this goal, a is computed by estimating how much TCP would get on each MPTCP path (this is easy, as round-trip time and loss-rates estimates are known) and ensuring that MPTCP in stable state gets at least that much. A detailed discussion on the design of the MPTCP congestion control algorithm is provided in [117].

For example, in the three-path example above, the flow will put 45% of its weight on each of the less congested path and 10% on the more congested path. This is intermediate between regular TCP (33% on each path) and a perfect load balancing algorithm (0% on the more congested path) that is impossible to implement in practice.

The window increase is capped at $1/w_r$, which ensures that the multipath flow can take no more capacity on either path than a single-path TCP flow would.

In setting the a parameter, Multipath TCP congestion control uses subflow delays to compute the target rate. Using delay for congestion control is not a novel idea: TCP Vegas [20], for instance, performs congestion control only by tracking RTT values. Vegas treats increasing RTTs as a sign of congestion, instead of relying on packet losses as regular TCP does. That is why Vegas loses out to regular TCP at shared bottlenecks, and is probably the reason for its lack of adoption. MPTCP does not treat delay as congestion: it just uses it to figure out the effective rate of a TCP connection on a given path. This allows MPTCP to compete fairly with TCP.

Alternative Congestion Controllers for Multipath TCP. The standardized Multipath TCP congestion control algorithm chooses a trade-off between load balancing, stability and the ability to quickly detect available capacity. The biggest contribution of this work is the clearly defined goals for what multipath congestion control should do, and an instantiation that achieves (most of) the stated goals in practice.

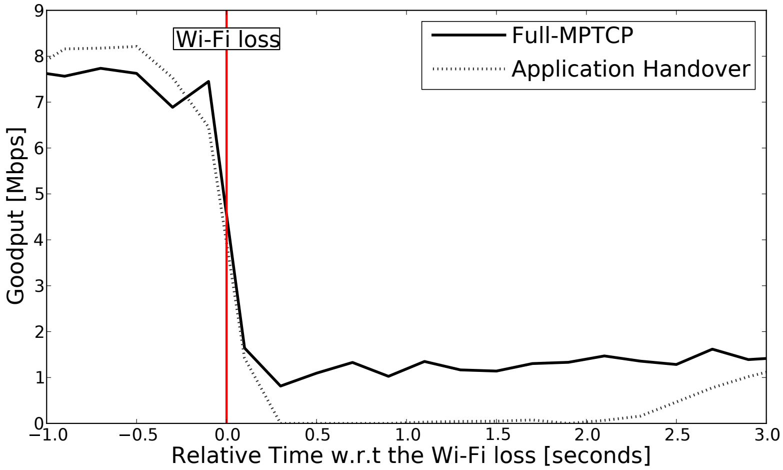

Figure 12: (Mobility) A mobile device is using both its WiFi and 3G interfaces, and then the WiFi interface fails. We plot the instantaneous throughputs of Multipath TCP and application-layer handover. (Reprinted from [80]; ©2012, ACM. Included here by permission.)

This research area is relatively new, and it is likely that more work will lead to better algorithms—if not generally applicable, then at least tailored to some practical use-cases. A new and interesting congestion controller called Opportunistic Linked Increases Algorithm (OLIA) has already been proposed [66] that offers better load balancing with seemingly few drawbacks.

We expect this area to be very active in the near future; of particular interest are designing multipath versions of high-speed congestion control variants deployed in practice, such as Cubic or Compound TCP.

4.3 Implementation and performance

We now briefly cover two of the most compelling use cases for Multipath TCP by showing a few evaluation results. We focus on mobile devices and datacenters but note that Multipath TCP can also help in other scenarios. For example, multi-homed web-servers can perform fine-grained load-balancing across their uplinks, while dual-stack hosts can use both IPv4 and IPv6 within a single Multipath TCP connection.

The full Multipath TCP protocol has been implemented in the Linux kernel; its congestion controller has also been implemented in the ns2 and htsim network simulators. The results presented in here are from the Linux kernel implementation [94].

The mobile measurements focus on a typical mode of operation where the device is connected to WiFi, the connection goes down and the phone switches to using 3G. The setup uses a Linux laptop connected to a WiFi and a 3G network, downloading a file using HTTP.

3G to Wifi handover is implemented in today's phones by changing the application to monitor the network interfaces. When the app detects the loss of the interface, it creates a new TCP connection to the server and the connection can resume. This solution, while simple, is not applicable to all applications because some of the bytes succesfully delivered by the old connection may be resent by the new one.

Applications that rely on HTTP GET requests (with no side effects) are, however, easy to change. The HTTP range header allows a client to resume the download of a file from a specified offset. This is the de-facto standard for today's apps.

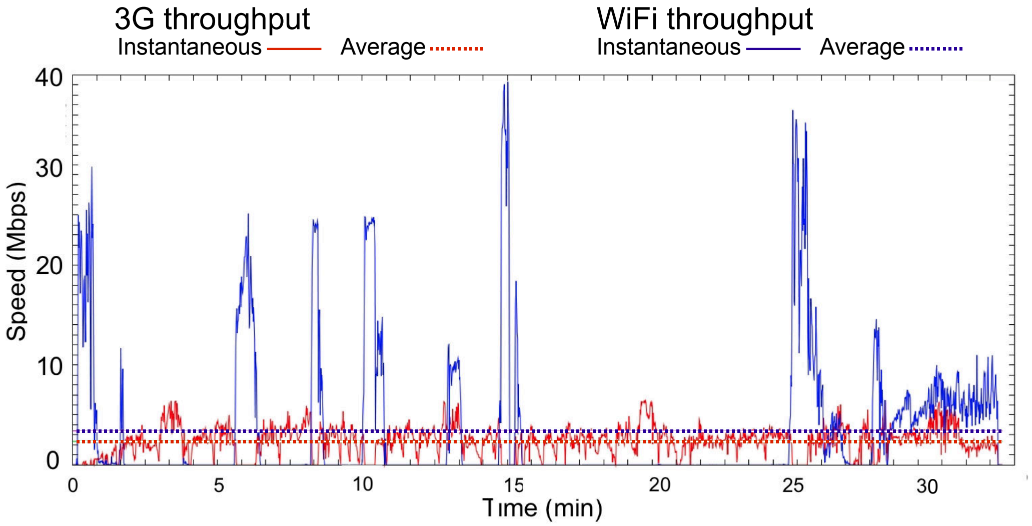

Figure 13: (3G to WiFi Handover) A Linux distribution is downloaded by a mobile user while on the Bucharest subway. 3G coverage is ubiquitous. WiFi is available only in stations.

Figure 12 compares application layer-handover (with HTTP-range) against Multipath TCP. The figure shows a smooth handover with Multipath TCP, as data keeps flowing despite the interface change. With application-layer handover there is a downtime of 4 seconds where the transfer stops-this is because it takes time for the application to detect the interface down event, and it takes time for 3G to ramp up.

Multipath TCP enables unmodified mobile applications to survive interface changes with little disruption. Selected apps can be modified today to support handover, but their performance is worse than with MPTCP. A more detailed discussion of the utilization of Multipath TCP in WiFi/3G environments may be found in [80].

We also present a real mobility trace in Figure 13, where a mobile user downloads a Linux distro on his laptop while travelling on the Bucharest underground. The laptop uses a 3G dongle to connect to the cellular network and its WiFi NIC to connect to access points available in stations.

The figure shows the download speeds of the two interfaces during a 30 minute underground ride. 3G throughput is stable: the average is 2.3Mbps (shown with a dotted line on the graph), and the instantaneous throughput varies inversely proportional with the distance to the 3G cell. WiFi throughput is much more bursty: in some stations the throughput soars to 40Mbps, while in others it is zero, as the laptop doesn't manage to associate and obtain an IP address quickly enough. The average WiFi throughput (3.3Mbps) is higher than the average 3G throughput.

While MPTCP uses both interfaces in this experiment, using just WiFi when it is available may be preferable as it reduces 3G data bills and reduces the load of the cellular network. This is easy to implement with MPTCP, as it supports a simple prioritization mechanism that allows the client to inform the server to send data via preferred subflow(s) [9].

[9] MPTCP sends an MP_PRIO option to inform the remote end about changes in priority for the subflows.

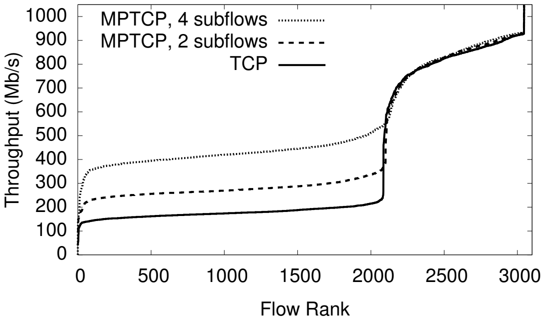

Figure 14: (Datacenter load-balancing) This graph compares standard TCP with MPTCP with two and four flows, when tested on an EC2 testbed with 40 instances. Each host uses iperf sequentially to all other hosts. We plot the performance of all flows (Y axis) in increasing order of their throughputs (X axis). (Reprinted from [92], ©ACM, 2011. Included here by permission)

Datacenters. We also show results from running Multipath TCP in a different scenario: the EC2 datacenter. Like most datacenters today, EC2 uses a redundant network topology where many paths are available between any pair of endpoints, and where connections are placed randomly onto available paths. In EC2, 40 machines (or instances) ran the Multipath TCP kernel. A simple experiment was run where every machine measured the throughput sequentially to every other machine using first TCP, then Multipath TCP with two and with four subflows. Figure 14 shows the sorted throughputs measured over 12 hours. The results show that Multipath TCP brings significant improvements compared to TCP in this scenario. Because the EC2 network is essentially a black-box , it is difficult to pinpoint the root cause for the improvements; however, a detailed analysis of the cases where Multipath TCP can help and why in is presented in [92].

4.4 Impact of Multipath TCP

MPTCP is deployable today, as it was designed and shown to work over existing networks and unmodified applications. Whether MPTCP will be adopted or not is unclear at this point; only time will tell. If MPTCP is adopted, however, it will likely affect the development of both the network and the applications running on top of it. Here we briefly speculate on what the impact might be. Multipath TCP embeds two key mechanisms: load balancing to avoid congested paths, and the ability to use different IP addresses in the same connection. Having these mechanisms implemented at the end-hosts means other layers (e.g. the network) may become simpler, and more efficient.

For instance, MPTCP seamlessly uses available bandwidth even if it is short-lived (e.g. WiFi in underground stations), and provides robustness when a subset of paths fail by quickly resending data on working paths. This could take some of the load imposed on BGP, that today must quickly reconverge in the case of a failure. With MPTCP at the endpoints failures would be “masked” automatically, allowing BGP to react to failures slowly which will ensure there are no oscillations. However, one must ensure that the paths used by the end-systems are disjoint and that the failure does not affect all of them.

Another example is layer 2 mobility, implemented in both WiFi and cellular networks, that aims to do a fast handover from one access point to the next, or from one cell to the next. With MPTCP as a transport protocol, a WiFi client could connect simultaneously to two access points or cells (provided the L2 protocol allows it), and load balance traffic on the best link. MPTCP could replace fast handovers with slow handovers that are more robust. Of course, some changes may be needed at layer 2 to support such load balancing.

Network topologies may be designed differently if MPTCP is the transport protocol. An example is GRIN [4], a work that proposes to change existing datacenter networks by randomly interconnecting servers in the same rack directly, using their free NIC ports. This allows a server to opportunistically send traffic at speeds larger than its access link by having its idle neighbor relay some traffic on its behalf. With a minor topology change and MPTCP, GRIN manages to better utilize the datacenter network core.

The mechanisms that allow MPTCP to use different IPs in the same transport connection (i.e. the connection identifier) also allow connections to be migrated across physical machines with different IP addresses. One direct application is seamless virtual machine migration: an MPTCP-enabled guest virtual machine can just resume its connections after it is migrated.

On the application side many optimizations are possible. Applications like video and audio streaming only need a certain bitrate to ensure user satisfaction - they rarely fully utilize the wireless capacity. Our mobility graph in the previous section shows that, in principle, WiFi has enough capacity to support these apps, even if it is only available in stations. A simple optimization would be to download as much of the video as possible while on WiFi, and only use 3G when the playout buffer falls below a certain threshold. This would ensure a smooth viewing experience while pushing as much traffic as possible over WiFi.

The examples shown in this section are only meant to be illustrative, and their potential impact or even feasibility is not clear at this point. Further, the list is not meant to be exhaustive. We have only provided it to show that Multipath TCP benefits go beyond than increasing throughput, and could shape both the lower and the upper layers of the protocol stack in the future.

5 Minion

As the Internet has grown and evolved over the past few decades, congestion control algorithms and extensions such as MPTCP continue to reflect an evolving TCP. However, a proliferation of middleboxes such as Network Address Translators (NATs), Firewalls, and Performance Enhancing Proxies (PEPs), has arguably stretched the waist of the Internet hourglass upwards from IP to include TCP and UDP [98, 45, 86], making it increasingly difficult to deploy new transports and to use anything but TCP and UDP on the Internet.

TCP [89] was originally designed to offer applications a convenient, high-level communication abstraction with semantics emulating Unix file I/O or pipes: a reliable, ordered bytestream, through an end-to-end channel (or connection). As the Internet has evolved, however, applications needed better abstractions from the transport. We start this section by examining how TCP's role in the network has evolved from a communication *abstraction* to a communication *substrate*, why its in-order delivery model makes TCP a poor substrate, why other OS-level transports have failed to replace TCP in this role, and why UDP is inadequate as the only alternative substrate to TCP.

5.1 Rise of Application-Level Transports

The transport layer's traditional role in a network stack is to build high-level communication abstractions convenient to applications, atop the network layer's basic packet delivery service. TCP's reliable, stream-oriented design [89] exemplified this principle, by offering an inter-host communication abstraction modeled on Unix pipes, which were the standard *intra-host* communication abstraction at the time of TCP's design. The Unix tradition of implementing TCP in the OS kernel offered further convenience, allowing much application code to ignore the difference between an open disk file, an intra-host pipe, or an inter-host TCP socket.

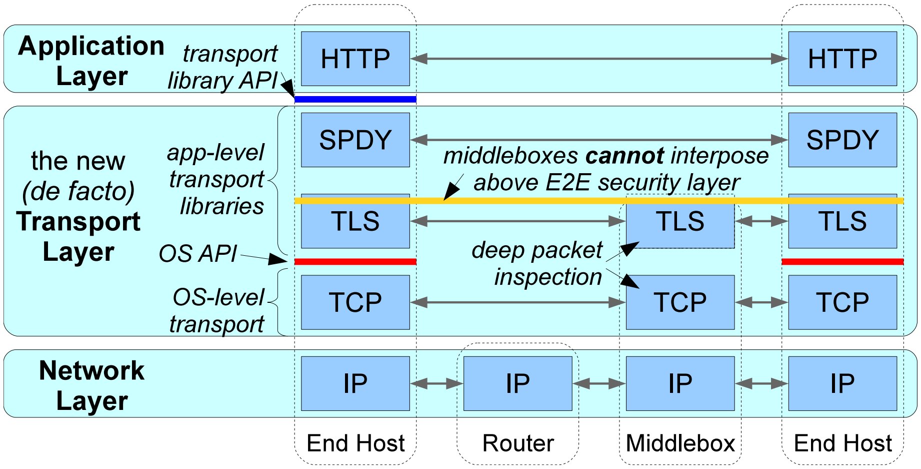

Figure 15: Today's "*de facto* transport layer" is effectively split between OS and application code. (Reprinted from [78]. Included here by permission.)

Instead of building directly atop traditional OS-level transports such as TCP or UDP, however, today's applications frequently introduce additional transport-like protocol layers at user-level, typically implemented via application-linked libraries. Examples include the ubiquitous SSL/TLS [35], media transports such as RTP [101], and experimental multi-streaming transports such as SST [44], SPDY [1], and ØMQ [2]. Applications increasingly use HTTP or HTTPS over TCP as a substrate [86]; this is also illustrated by the W3C's WebSocket interface [114], which offers general bidirectional communication between browser-based applications and Web servers atop HTTP and HTTPS.

In this increasingly common design pattern, the "transport layer" as a whole has in effect become a stack of protocols straddling the OS-application boundary. Figure 15 illustrates one example stack, representing Google's experimental Chrome browser, which inserts SPDY for multi-streaming and TLS for security at application level, atop the OS-level TCP.

One can debate whether a given application-level protocol fits some definition of "transport" functionality. The important point, however, is that *today's applications no longer need, or expect, the underlying OS to provide "convenient" communication abstractions: an application simply links in libraries, frameworks, or middleware offering the abstractions it desires.* What today's applications need from the OS is not convenience, but *an efficient substrate* atop which application-level libraries can build the desired abstractions.

5.2 TCP's Latency Tax

While TCP has proven to be a popular substrate for application-level transports, using TCP in this role converts its delivery model from a blessing into a curse. Application-level transports are just as capable as the kernel of sequencing and reassembling packets into a logical data unit or "frame" [29]. By delaying any segment's delivery to the application until all prior segments are received and delivered, however, TCP imposes a "latency tax" on all segments arriving within one round-trip time (RTT) after any single lost segment.

This latency tax is a fundamental byproduct of TCP's in-order delivery model, and is irreducible, in that an application-level transport cannot "claw back" the time a potentially useful segment has wasted in TCP's buffers. The best the application can do is simply to *expect* higher latencies to be common. A conferencing application can use a longer jitter buffer, for example, at the cost of increasing user-perceptible lag. Network hardware advances are unlikely to address this issue, since TCP's latency tax depends on RTT, which is lower-bounded by the speed of light for long-distance communications.

5.3 Alternative OS-level Transports

All standardized OS-level transports since TCP, including UDP [87], RDP [112], DCCP [67], and SCTP [105], support out-of-order delivery. The Internet's evolution has created strong barriers against the widespread deployment of new transports other than the original TCP and UDP, however. These barriers are detailed elsewhere [98, 45, 86], but we summarize two key issues here.

First, adding or enhancing a "native" transport built atop IP involves modifying popular OSes, effectively increasing the bar for widespread deployment and making it more difficult to evolve transport functionality below the red line representing the OS API in Figure 15. Second, the Internet's original "dumb network" design, in which routers that "see" only up to the IP layer, has evolved into a "smart network" in which pervasive middleboxes perform deep packet inspection and interposition in transport and higher layers. Firewalls tend to block "anything unfamiliar" for security reasons, and Network Address Translators (NATs) rewrite the port number in the transport header, making both incapable of allowing traffic from a new transport without explicit support for that transport. Any packet content not protected by end-to-end security such as TLS—the yellow line in Figure 15—has become "fair game" for middleboxes to inspect and interpose on [95], making it more difficult to evolve transport functionality anywhere below that line.

5.4 Why Not UDP?

As the only widely-supported transport with out-of-order delivery, UDP offers a natural substrate for application-level transports. Even applications otherwise well-suited to UDP's delivery model often favor TCP as a substrate, however.

A recent study found over 70% of streaming media using TCP [50], and even latency-sensitive conferencing applications such as Skype often use TCP [12]. Firewalls can monitor the TCP state machine to help thwart network attacks [54], but determining an application session's state atop UDP requires application-specific analysis [77], creating an incentive for firewalls to block UDP in general.

In general, network middleboxes support UDP widely but not *universally*. For this reason, latency-sensitive applications seeking maximal connectivity "in the wild" often fall back to TCP when UDP connectivity fails. Skype [12] and Microsoft's DirectAccess VPN [31], for example, support UDP but can masquerade as HTTP or HTTPS streams atop TCP when required for connectivity.

TCP can offer performance advantages over UDP as well. For applications requiring congestion control, an OS-level implementation in TCP may be more timing-accurate than an application-level implementation in a UDP-based protocol, because the OS kernel can avoid the timing artifacts of system calls and process scheduling [118]. Hardware TCP offload engines can optimize common-case efficiency in end hosts [76], and performance enhancing proxies can optimize TCP throughput across diverse networks [22, 26]. Since middleboxes can track TCP's state machine, they impose much longer idle timeouts on open TCP connections—nominally two hours [49]—whereas UDP-based applications must send keepalives every two minutes to keep an idle connection open [9], draining power on mobile devices.

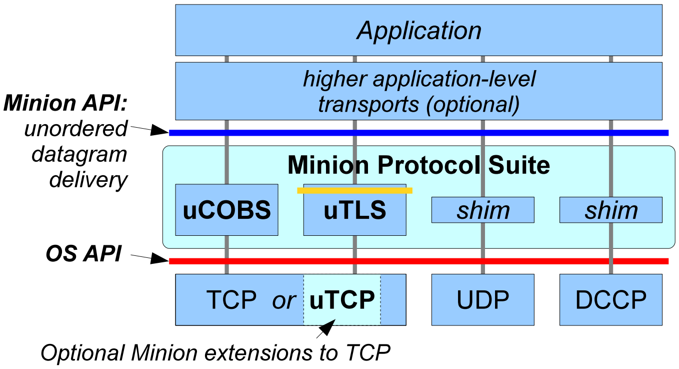
Application
higher application-level transports (optional)
Minion API: unordered datagram delivery
Minion Protocol Suite
uCOBS
uTLS
shim
shim
OS API
TCP or uTCP
UDP
DCCP
Optional Minion extensions to TCP

Figure 16: Minion architecture (Reprinted from [78]. Included here by permission.)

5.5 Breaking Out Of the Transport Logjam

Applications and application developers care most about services that the networking infrastructure offers to them and not how packets look on the wire; that is, they care about new transport *services*, not new transport *protocols*. On the other hand, middleboxes care most about how packets look on the wire, and generally do not care about what services are offered to the applications; that is, changing the transport protocol's bits on the wire will require changing middleboxes to respond to these changes as well.

For application developers, TCP versus UDP represents an "all-or-nothing" choice on the spectrum of services applications need. Applications desiring some but not all of TCP's services, such as congestion control but unordered delivery, must reimplement and tune all other services atop UDP or suffer TCP's performance penalties.

Without dismissing UDP's usefulness as a truly "least-common-denominator" substrate, we believe the factors above suggest that TCP will also remain a popular substrate—even for latency-sensitive applications that can benefit from out-of-order delivery—and that a deployable, backward-compatible workaround to TCP's latency tax can significantly benefit such applications.

Recognizing that the development of application-level transports and the use of TCP as a substrate under them is likely to continue and expand, we now describe *Minion*, an architecture for efficient but backward-compatible unordered delivery over extant transports, including TCP. Minion consists of $uTCP$, a small OS extension adding basic unordered delivery primitives to TCP, and two application-level protocols implementing datagram-oriented delivery services that function on either uTCP or unmodified TCP stacks.

While building a new transport on UDP or using alternative OS-level transports where available are both perfectly reasonable design approaches, Minion offers a uniform interface for applications to use along with an alternative option where the ubiquitous TCP protocol is adequate, but the sockets API is not; where a new transport service can be offered using TCP's bits on the wire.

5.6 Minion Architecture Overview

Minion is an architecture and protocol suite designed to provide efficient unordered delivery built atop existing transports. Minion itself offers no high-level abstractions: its goal is to serve applications and higher application-level transports, by acting as a "packhorse" carrying raw datagrams as reliably and efficiently as possible across today's diverse and change-averse Internet.

Figure 16 illustrates Minion's architecture. Applications and higher application-level transports link in and use Minion in the same way as they already use existing application-level transports such as DTLS [97], the datagram-oriented analog of SSL/TLS [35]. In contrast with DTLS's goal of layering security atop datagram transports such as UDP or DCCP, Minion's goal is to offer efficient datagram delivery atop *any* available OS-level substrate, including TCP.

Minion consists of several application-level transport protocols, together with a set of optional enhancements to end hosts' OS-level TCP implementations.

Minion's enhanced OS-level TCP stack, called *uTCP* ("unordered TCP"), includes sender- and receiver-side API features supporting unordered delivery and prioritization, as detailed later in this section. These enhancements affect only the OS API through which application-level transports such as Minion interact with the TCP stack, and make *no* changes to TCP's wire protocol.

Minion's application-level protocol suite currently consists of the following main components:

- u**COBS** is a protocol that implements a minimal unordered datagram delivery service atop either unmodified TCP or uTCP, using *Consistent-Overhead Byte Stuffing*, or COBS encoding [23] to facilitate out-of-order datagram delimiting and prioritized delivery, as described later in Section 5.8.3.
- u**TLS** is a modification of the traditionally stream-oriented TLS [35], offering a secure, unordered datagram delivery service atop TCP or uTCP. The wire-encoding of uTLS streams is designed to be indistinguishable in the network from conventional, encrypted TLS-over-TCP streams (e.g., HTTPS), offering a maximally conservative design point that makes no network-visible changes "below the yellow line" in Figure 16. Section 5.8.3 describes uTLS.
- Minion adds shim layers atop OS-level datagram transports, such as UDP and DCCP, to offer applications a consistent API for unordered delivery across multiple OS-level transports. Since these shims are merely wrappers for OS transports already offering unordered delivery, this paper does not discuss them in detail.

Minion currently leaves to the application the decision of *which* protocol to use for a given connection: e.g., uCOBS or uTLS atop TCP/uTCP, or OS-level UDP or DCCP via Minion's shims. Other ongoing work explores *negotiation protocols* to explore the protocol configuration space dynamically, optimizing protocol selection and configuration for the application's needs and the network's constraints [46]. Many applications already incorporate simple negotiation schemes, however—e.g., attempting a UDP connection first and falling back to TCP if that fails—and adapting these mechanisms to engage Minion's protocols according to application-defined preferences and decision criteria should be straightforward.

5.7 Compatibility and Deployability

Minion addresses the key barriers to transport evolution, outlined in Section 5.3, by creating a backward-compatible, incrementally deployable substrate for new application-layer transports desiring unordered delivery. Minion's deployability rests on the fact that it can, when necessary, avoid relying on changes either "below the red line" in the end hosts (the OS API in Figure 16), or "below the yellow line" in the network (the end-to-end security layer in Figure 16).

While Minion's uCOBS and uTLS protocols offer maximum performance benefits from out-of-order delivery when both endpoints include OS support for Minion's uTCP enhancements, uCOBS and uTLS still function and interoperate correctly even if neither endpoint supports uTCP, and the application need not know or care whether the underlying OS supports uTCP. If only one endpoint OS supports uTCP, Minion still offers incremental performance benefits, since uTCP's sender-side and receiver-side enhancements are independent. A uCOBS or uTLS connection atop a mixed TCP/uTCP endpoint-pair benefits from uTCP's sender-side enhancements for datagrams sent by the uTCP endpoint, and the connection benefits from uTCP's receiver-side enhancements for datagrams arriving at the uTCP host.

Addressing the challenge of network-compatibility with middleboxes that filter new OS-level transports and sometimes UDP, Minion offers application-level transports a continuum of substrates representing different tradeoffs between suitability to the application's needs and compatibility with the network.

An application can use unordered OS-level transports such as UDP, DCCP [67], or SCTP [105], for paths on which they operate, but Minion offers an unordered delivery alternative wire-compatible not only with TCP, but with the ubiquitous TLS-over-TCP streams on which HTTPS (and hence Web security and E-commerce) are based, likely to operate in almost any network environment purporting to offer "Internet access."

5.8 Minion's Unordered Delivery using TCP and TLS

We now briefly discuss Minion's true unordered delivery over bytestream protocols; we describe how Minion provides true unordered datagram delivery without modifying the TCP and TLS wire-formats.

Minion enhances the OS's TCP stack with API enhancements supporting unordered delivery in both TCP's send and receive paths, enabling applications to reduce transmission latency at both the sender- and receiver-side end hosts when both endpoints support uTCP. Since uTCP makes no change to TCP's wire protocol, two endpoints need not "agree" on whether to use uTCP: one endpoint gains latency benefits from uTCP even if the other endpoint does not support it. Further, an OS may choose independently whether to support the sender- and receiver-side enhancements, and when available, applications can activate them independently.

uTCP does *not* seek to offer "convenient" or "clean" unordered delivery abstractions directly at the OS API. Instead, uTCP's design is motivated by the goals of maintaining exact compatibility with TCP's existing wire-visible protocol and behavior, and facilitating deployability by minimizing the extent and complexity of changes to the OS's TCP stack.

5.8.1 uTCP: Receiver-Side Modifications

A conventional TCP receiver delivers data in-order to the receiving application, holding back any data that is received out of order. uTCP modifies the TCP receive path, enabling a receiving application to request immediate delivery of data that is received by uTCP, both in order and out of order.

uTCP makes two modifications to a conventional TCP receiver. First, whereas a conventional TCP stack delivers received data to the application only when prior gaps in the TCP sequence space are filled, the uTCP receiver makes data segments available to the application immediately upon receipt, skipping TCP's usual reordering queue. The data the uTCP stack delivers to the application in successive application reads may skip forward and backward in the transmitted byte stream, and uTCP may even deliver portions of the transmitted stream multiple times. uTCP guarantees only that the data returned by each application read corresponds to *some* contiguous sequence of bytes in the sender's transmitted stream.

Second, when servicing an application's read, the uTCP receiver also delivers the logical offset of the first returned byte in the sender's original byte stream—information that a TCP receiver must maintain to arrange received segments in order.

Figure 17 illustrates uTCP's receive-side behavior, in a simple scenario where three TCP segments arrive in succession: first an in-order segment, then an out-of-order segment, and finally a segment filling the gap

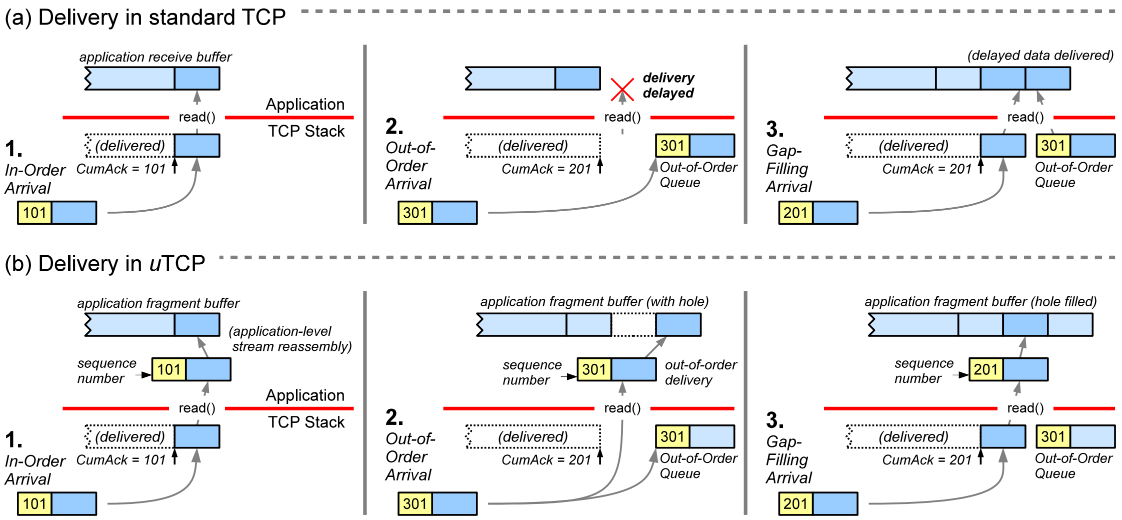

Figure 17: Delivery behavior of (a) standard TCP, and (b) uTCP, upon receipt of in-order and out-of-order segments. (Reprinted from [78]. Included here by permission.)

between the first two. With uTCP, the application receives each segment as soon as it arrives, along with the sequence number information it needs to reconstruct a complete internal view of whichever fragments of the TCP stream have arrived.

5.8.2 uTCP: Sender-Side Modifications

While uTCP's receiver-side enhancements address the "latency tax" on segments waiting in TCP's reordering buffer, TCP's sender-side queue can also introduce latency, as segments the application has already written to a TCP socket—and hence "committed" to the network—wait until TCP's flow and congestion control allow their transmission. Many applications can benefit from the ability to "late-bind" their decision on *what* to send until the last possible moment, and also from being able to transmit a message of higher priority that bypasses any lower priority messages in the sender-side queue.

A uTCP sender allows a sending application to specify a tag with each application write, which the uTCP sender currently interprets as a priority level. Instead of unconditionally placing the newly-written data at the tail of the send queue as TCP normally would, uTCP *inserts* the newly-written data into the send queue just *before* any lower-priority data in the send queue not yet transmitted.

With these modifications to a TCP stack, none of which require changes to the TCP wire-format, uTCP offers an interface which, while not convenient for applications, is powerful. In the next Section, we discuss how we build a userspace library that uses this interface that provides a simple unordered delivery service, unordered delivery of encrypted messages, and logically separate data streams within a single uTCP connection.

5.8.3 Datagrams atop uTCP

Applications built on datagram substrates such as UDP generally assume the underlying layer preserves datagram boundaries. TCP's stream-oriented semantics do not preserve any application-relevant frame boundaries within a stream, however. Both the TCP sender and network middleboxes can and do coalesce TCP segments or re-segment TCP streams in unpredictable ways [58].

Atop uTCP, a userspace library can reconstruct contiguous fragments in the received data stream using the metadata sequence number information that uTCP passes along at the receiver. However, providing unordered message delivery service atop uTCP requires delimiting application messages in the bytestream. While record delimiting is commonly done by application protocols such as HTTP, SIP, and many others, a key property that we require to provide a true unordered delivery service is that a receiver must be able to extract a given message independently of other messages. That is, as soon as a complete message is received, the message delimiting mechanism must allow for extraction of the message from the bytestream fragment, without relying on the receipt of earlier messages.

Minion implements self-delimiting messages in two ways:

1. To encode application datagrams efficiently, the userspace library employs *Consistent-Overhead Byte Stuffing*, or COBS [23] to delimit and extract messages. COBS is a binary encoding which eliminates *exactly* one byte value from a record's encoding with minimal bandwidth overhead. To encode an application record, COBS first scans the record for *runs* of contiguous marker-free data followed by exactly one marker byte. COBS then removes the trailing marker, instead *prepending* a non-marker byte indicating the run length. A special run-length value indicates a run of 254 bytes *not* followed by a marker in the original data, enabling COBS to divide arbitrary-length runs into 254-byte runs encoded into 255 bytes each, yielding a worst-case expansion of only 0.4%.

2. The userspace library coaxes out-of-order delivery from the *existing* TCP-oriented TLS wire format, producing an encrypted datagram substrate indistinguishable on the wire from standard TLS connections. TLS [35] already breaks its communication into *records*, encrypts and authenticates each record, and prepends a header for transmission on the underlying TCP stream. TLS was designed to decrypt records strictly in-order, however, creating challenges which the userspace library overcomes [78]. Run on port 443, our encrypted stream atop uTCP is indistinguishable from HTTPS—regardless of whether the application actually uses HTTP headers, since the HTTP portion of HTTPS streams are TLS-encrypted anyway. Deployed this way, Minion effectively offers an end-to-end protected substrate in the "HTTP as the new narrow waist" philosophy [86].

5.9 Impact on Real Applications

Minion's unordered delivery service benefits a number of applications; we refer the reader to detailed discussion and experiments in [78]. Of these applications, we briefly discuss how Minion fits within ongoing efforts to develop a next-generation transport for the web, such as SPDY[1] and the Internet Engineering Task Force (IETF)'s HTTP/2.0 (httpbis) effort. Developing a next-generation HTTP requires either submitting to TCP's latency tax for backward compatibility, as with SPDY's use of TLS/TCP, or developing and deploying new transports atop UDP, neither of which, as we discussed earlier in this section, is a satisfying alternative.

Minion bridges this gap and demonstrates that it is possible to obtain unordered delivery from wire-compatible TCP and TLS streams with surprisingly small changes to TCP stacks and application-level code. These protocols offer latency-sensitive applications performance benefits comparable to UDP or DCCP, with the compatibility benefits of TCP and TLS. Without discounting the value of UDP and newer OS-level transports, Minion offers a more conservative path toward the performance benefits of unordered delivery, which we expect to be useful to applications that use TCP for a variety of pragmatic reasons.

6 Conclusion

The Transport Layer in the Internet evolved for nearly two decades, but it has been stuck for over a decade now. A proliferation of middleboxes in the Internet, devices in the network that look past the IP header, has shifted the waist of the Internet hourglass upward from IP to include UDP and TCP, the legacy workhorses of the Internet. While popular for many different reasons, middleboxes thus deviate from the Internets end-to-end design, creating large deployment black-holes—singularities where legacy transports get through, but any new transport technology or protocol fails, severely limiting transport protocol evolution. The fallout of this ossification is that new transport protocols, such as SCTP and DCCP, that were developed to offer much needed richer end-to-end services to applications, have had trouble getting deployed since they require changes to extant middleboxes.

Multipath TCP is perhaps the most significant change to TCP in the past twenty years. It allows existing TCP applications to achieve better performance and robustness over today's networks, and it has been standardized at the IETF. The Linux kernel implementation shows that these benefits can be obtained in practice. However, as with any change to TCP, the deployment bar for Multipath TCP is very high: only time will tell whether the benefits it brings will outweigh the added complexity it brings in the end-host stacks.

The design of Multipath TCP has been a lengthy, painful process that took around five years. Most of the difficulty came from the need to support existing middlebox behaviors, while offering the exact same service to applications as TCP. Although the design space seemed wide open in the beginning, in the end we were *just* able to evolve TCP this way: for many of the design choices there was only one viable option that could be used. When the next major TCP extension is designed in a network with even more middleboxes, will we, as a community, be as lucky?

A pragmatic answer to the inability to deploy new transport protocols is Minion. It allows deploying new transport services by being backward compatible with middleboxes by encapsulating new protocols inside TCP. Minion demonstrates that it is possible to obtain unordered delivery and multistreaming from wire-compatible TCP and TLS streams with surprisingly small changes to TCP stacks and application-level code. Minion offers a path toward the performance benefits of unordered delivery, which we expect to be useful to applications that use TCP for a variety of pragmatic reasons.

Early in the Internet's history, all IP packets could travel freely through the Internet, as IP was the narrow waist of the protocol stack. Eventually, apps started using UDP and TCP exclusively, and some, such as Skype, used them adaptively, probably due to security concerns in addition to the increasing proliferation of middleboxes that allowed only UDP and TCP through. As a result, UDP and TCP over IP were then perceived to constitute the new waist of the Internet. (We'll note that HTTP has also recently been suggested as the new waist [86].)

Our observation is that whatever the new waist is, middleboxes will embrace it and optimize for it: if MPTCP and/or Minion become popular, it is likely that middleboxes will be devised that understand these protocols to optimize for the most successful use-case of these protocols and to help protect any vulnerable applications using them. One immediate answer from an application would be to use the encrypted communication proposed in Minion—but actively hiding information from a network operator can potentially encourage the network operator to embed middleboxes that intercept encrypted connections, effectively mounting man-in-the-middle attacks to control traffic over their network, as is already being done in several corporate firewalls [72]. To bypass these middleboxes, new applications may encapsulate their data *even* deeper, leading to a vicious circle resembling an "arms race" for control over network use.

This "arms race" is a symptom of a fundamental tussle between end-hosts and the network: end-hosts will always want to deploy new applications and services, while the network will always want to allow and optimize only existing ones [28]. To break out of this vicious circle, we propose that end-hosts and the network must co-operate, and that they must build cooperation into their protocols. Designing and providing protocols and incentives for this cooperation may hold the key to creating a truly evolvable transport (and Internet) architecture.

Acknowledgements

We would like to thank the reviewers whose comments have improved this chapter. We would also like to thank Adrian Vladulescu for the measurements presented in figure 13.

References

[1] SPDY: An Experimental Protocol For a Faster Web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[2] ZeroMQ: The intelligent transport layer. `http://www.zeromq.org`.

[3] AFANASYEV, A., TILLEY, N., REIHER, P., AND KLEINROCK, L. Host-to-Host Congestion Control for TCP. *IEEE Communications Surveys & Tutorials 12*, 3 (2012), 304–342.

[4] AGACHE, A., AND RAICIU, C. GRIN: utilizing the empty half of full bisection networks. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association, pp. 7–7.

[5] ALLMAN, M. Comments on selecting ephemeral ports. *SIGCOMM Comput. Commun. Rev. 39*, 2 (Mar. 2009), 13–19.

[6] ALLMAN, M., FLOYD, S., AND PARTRIDGE, C. Increasing TCP's Initial Window. RFC 3390 (Proposed Standard), Oct. 2002.

[7] ALLMAN, M., OSTERMANN, S., AND METZ, C. FTP Extensions for IPv6 and NATs. RFC 2428 (Proposed Standard), Sept. 1998.

[8] ALLMAN, M., PAXSON, V., AND BLANTON, E. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[9] AUDET, F., AND JENNINGS, C. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), Jan. 2007.

[10] BAKER, F. Requirements for IP Version 4 Routers. RFC 1812 (Proposed Standard), June 1995.

[11] BALAKRISHNAN, H., RAHUL, H., AND SESHAN, S. An integrated congestion management architecture for internet hosts. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1999), SIGCOMM '99, ACM, pp. 175–187.

[12] BASET, S. A., AND SCHULZRINNE, H. An analysis of the Skype peer-to-peer Internet telephony protocol. In *IEEE INFOCOM* (Apr. 2006).

[13] BEGEN, A., WING, D., AND CAENEGEM, T. V. Port Mapping between Unicast and Multicast RTP Sessions. RFC 6284 (Proposed Standard), June 2011.

[14] BEVERLY, R., BERGER, A., HYUN, Y., AND CLAFFY, K. Understanding the efficacy of deployed internet source address validation filtering. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 356–369.

[15] BIRRELL, A., AND NELSON, B. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.

[16] BONAVENTURE, O. *Computer Networking : Principles, Protocols and Practice*. Saylor foundation, 2012. Available from `http://inl.info.ucl.ac.be/cnp3`.

[17] BONAVENTURE, O., HANDLEY, M., AND RAICIU, C. An Overview of Multipath TCP. *Usenix ;login: magazine 37*, 5 (Oct. 2012).

[18] BRADEN, R. Requirements for Internet Hosts - Communication Layers. RFC 1122 (INTERNET STANDARD), Oct. 1989.

[19] BRADEN, R. T/TCP – TCP Extensions for Transactions Functional Specification. RFC 1644 (Historic), July 1994.

[20] BRAKMO, L., O'MALLEY, S., AND PETERSON, L. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications* (New York, NY, USA, 1994), SIGCOMM '94, ACM, pp. 24–35.

[21] BUDZISZ, L., GARCIA, J., BRUNSTROM, A., AND FERRÚS, R. A taxonomy and survey of SCTP research. *ACM Computing Surveys 44*, 4 (Aug. 2012), 1–36.

[22] CARPENTER, B., AND BRIM, S. Middleboxes: Taxonomy and Issues. RFC 3234 (Informational), Feb. 2002.

[23] CHESHIRE, S., AND BAKER, M. Consistent overhead byte stuffing. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1997), SIGCOMM '97, ACM, pp. 209–220.

[24] CHU, J. Tuning TCP Parameters for the 21st Century. Presented at IETF75, July 2009.

[25] CHU, J., DUKKIPATI, N., CHENG, Y., AND MATHIS, M. Increasing TCP's Initial Window. Internet draft, draft-ietf-tcpm-initcwnd, work in progress, February 2013.

[26] CISCO. Rate-Based Satellite Control Protocol, 2006.

[27] CLARK, D. The design philosophy of the darpa internet protocols. In *Symposium proceedings on Communications architectures and protocols* (New York, NY, USA, 1988), SIGCOMM '88, ACM, pp. 106–114.

[28] CLARK, D., WROCLAWSKI, J., SOLLINS, K., AND BRADEN, R. Tussle in cyberspace: defining tomorrow's internet. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2002), SIGCOMM '02, ACM, pp. 347–356.

[29] CLARK, D. D., AND TENNENHOUSE, D. L. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM symposium on Communications architectures & protocols* (New York, NY, USA, 1990), SIGCOMM '90, ACM, pp. 200–208.

[30] D'Acunto, L., Pouwelse, J., and Sips., H. A Measurement of NAT and Firewall Characteristics in Peer-to-Peer Systems. In *Proceedings of the ASCI Conference* (2009).

[31] Davies, J. DirectAccess and the thin edge network. *Microsoft TechNet Magazine* (May 2009).

[32] de Vivo, M., de Vivo, G., Koeneke, R., and Isern, G. Internet vulnerabilities related to TCP/IP and T/TCP. *SIGCOMM Comput. Commun. Rev. 29*, 1 (Jan. 1999), 81–85.

[33] Detal, G. tracebox. http://www.tracebox.org.

[34] Dhamdhere, A., Luckie, M., Huffaker, B., claffy, k., Elmokashfi, A., and Aben, E. Measuring the deployment of IPv6: topology, routing and performance. In *Proceedings of the 2012 ACM conference on Internet measurement conference* (New York, NY, USA, 2012), IMC '12, ACM, pp. 537–550.

[35] Dierks, T., and Rescorla, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008.

[36] Duke, M., Braden, R., Eddy, W., and Blanton, E. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614 (Informational), Sept. 2006.

[37] Eddy, W. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational), Aug. 2007.

[38] Egevang, K., and Francis, P. The IP Network Address Translator (NAT). RFC 1631 (Informational), May 1994.

[39] Faber, T., Touch, J., and Yue, W. The TIME-WAIT state in TCP and its effect on busy servers. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (1999), vol. 3, IEEE, pp. 1573–1583.

[40] Fall, K., and Stevens, R. *TCP/IP Illustrated, Volume 1: The Protocols*, vol. 1. Addison-Wesley Professional, 2011.

[41] Ferguson, P., and Senie, D. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice), May 2000.

[42] Fonseca, R., Porter, G., Katz, R., Shenker, S., and Stoica, I. IP options are not an option. Tech. Rep. UCB/EECS-2005-24, UC Berkeley, Berkeley, CA, 2005.

[43] Ford, A., Raiciu, C., Handley, M., and Bonaventure, O. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), Jan. 2013.

[44] Ford, B. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 361–372.

[45] Ford, B., and Iyengar, J. Breaking up the transport logjam. In *7th Workshop on Hot Topics in Networks (HotNets-VII)* (Oct. 2008).

[46] Ford, B., and Iyengar, J. Efficient cross-layer negotiation. In *8th Workshop on Hot Topics in Networks (HotNets-VIII)* (Oct. 2009).

[47] GONT, F. Survey of Security Hardening Methods for Transmission Control Protocol (TCP) Implementations. Internet draft, draft-ietf-tcpm-tcp-security, work in progress, March 2012.

[48] GONT, F., AND BELLOVIN, S. Defending against Sequence Number Attacks. RFC 6528 (Proposed Standard), Feb. 2012.

[49] GUHA, S., BISWAS, K., FORD, B., SIVAKUMAR, S., AND SRISURESH, P. NAT Behavioral Requirements for TCP. RFC 5382 (Best Current Practice), Oct. 2008.

[50] GUO, L., TAN, E., CHEN, S., XIAO, Z., SPATSCHECK, O., AND ZHANG, X. Delving into internet streaming media delivery: a quality and resource utilization perspective. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), IMC '06, ACM, pp. 217–230.

[51] HAIN, T. Architectural Implications of NAT. RFC 2993 (Informational), Nov. 2000.

[52] HAN, H., SHAKKOTTAI, S., HOLLOT, C., SRIKANT, R., AND TOWSLEY, D. Multi-path TCP: a joint congestion control and routing scheme to exploit path diversity in the Internet. *IEEE/ACM Trans. Networking 14*, 6 (2006).

[53] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium* (2001), pp. 9–9.

[54] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-end Protocol Semantics. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2001), USENIX Association, pp. 9–9.

[55] HAYES, D., BUT, J., AND ARMITAGE, G. Issues with network address translation for SCTP. *SIGCOMM Comput. Commun. Rev. 39*, 1 (Dec. 2008), 23–33.

[56] HESMANS, B. Click elements to model middleboxes. `https://bitbucket.org/bhesmans/click`.

[57] HOLDREGE, M., AND SRISURESH, P. Protocol Complications with the IP Network Address Translator. RFC 3027 (Informational), Jan. 2001.

[58] HONDA, M., NISHIDA, Y., RAICIU, C., GREENHALGH, A., HANDLEY, M., AND TOKUDA, H. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 181–194.

[59] HUITEMA, C. Multi-homed TCP. Internet draft, work in progress, 1995.

[60] IREN, S., AMER, P., AND CONRAD, P. The transport layer: tutorial and survey. *ACM Comput. Surv. 31*, 4 (Dec. 1999), 360–404.

[61] IYENGAR, J., FORD, B., AILAWADI, D., AMIN, S. O., NOWLAN, M., TIWARI, N., AND WISE, J. Minion—an All-Terrain Packet Packhorse to Jump-Start Stalled Internet Transports. In *PFLDNeT 2010* (Nov. 2010).

[62] JACOBSON, V. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and protocols* (New York, NY, USA, 1988), SIGCOMM '88, ACM, pp. 314–329.

[63] Jacobson, V., Braden, R., and Borman, D. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[64] Jiang, S., Guo, D., and Carpenter, B. An Incremental Carrier-Grade NAT (CGN) for IPv6 Transition. RFC 6264 (Informational), June 2011.

[65] Kelly, F., and Voice, T. Stability of end-to-end algorithms for joint routing and rate control. *SIGCOMM Comput. Commun. Rev. 35*, 2 (Apr. 2005), 5–12.

[66] Khalili, R., Gast, N., Popovic, M., Upadhyay, U., and Le Boudec, J.-Y. MPTCP is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (New York, NY, USA, 2012), CoNEXT '12, ACM, pp. 1–12.

[67] Kohler, E., Handley, M., and Floyd, S. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), Mar. 2006.

[68] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, F. The click modular router. *ACM Trans. Comput. Syst. 18*, 3 (Aug. 2000), 263–297.

[69] Larsen, M., and Gont, F. Recommendations for Transport-Protocol Port Randomization. RFC 6056 (Best Current Practice), Jan. 2011.

[70] Larzon, L.-A., Degermark, M., Pink, S., Jonsson, L.-E., and Fairhurst, G. The Lightweight User Datagram Protocol (UDP-Lite). RFC 3828 (Proposed Standard), July 2004.

[71] Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and Jones, L. SOCKS Protocol Version 5. RFC 1928 (Proposed Standard), Mar. 1996.

[72] Marko, K. Using SSL Proxies To Block Unauthorized SSL VPNs. *Processor Magazine, www.processor.com 32*, 16 (July 2010), 23.

[73] Mathis, M., and Heffner, J. Packetization Layer Path MTU Discovery. RFC 4821 (Proposed Standard), Mar. 2007.

[74] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), Oct. 1996.

[75] McLaggan, D. Web Cache Communication Protocol V2, Revision 1. Internet draft, draft-mclaggan-wccp-v2rev1, work in progress, August 2012.

[76] Mogul, J. TCP offload is a dumb idea whose time has come. In *HotOS IX* (May 2003).

[77] Northcutt, S., Zeltser, L., Winters, S., Kent, K., and Ritchey, R. *Inside Network Perimeter Security*. SAMS Publishing, 2005.

[78] Nowlan, M., Tiwari, N., Iyengar, J., Amin, S. O., and Ford, B. Fitting square pegs through round pipes: Unordered delivery wire-compatible with TCP and TLS. In *NSDI* (Apr. 2012), vol. 12.

[79] Ong, L., and Yoakum, J. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May 2002.

[80] Paasch, C., Detal, G., Duchene, F., Raiciu, C., and Bonaventure, O. Exploring mobile/WiFi handover with multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design* (New York, NY, USA, 2012), CellNet '12, ACM, pp. 31–36.

[81] Paxson, V., and Allman, M. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), Nov. 2000.

[82] Paxson, V., Allman, M., Chu, J., and Sargent, M. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[83] Perreault, S., Yamagata, I., Miyakawa, S., Nakagawa, A., and Ashida, H. Common Requirements for Carrier-Grade NATs (CGNs). RFC 6888 (Best Current Practice), Apr. 2013.

[84] Pfeiffer, R. Measuring TCP Congestion Windows. Linux Gazette, March 2007.

[85] Phelan, T., Fairhurst, G., and Perkins, C. DCCP-UDP: A Datagram Congestion Control Protocol UDP Encapsulation for NAT Traversal. RFC 6773 (Proposed Standard), Nov. 2012.

[86] Popa, L., Ghodsi, A., and Stoica, I. Http as the narrow waist of the future internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (New York, NY, USA, 2010), Hotnets-IX, ACM, pp. 6:1–6:6.

[87] Postel, J. User Datagram Protocol. RFC 768 (INTERNET STANDARD), Aug. 1980.

[88] Postel, J. Internet Protocol. RFC 791 (INTERNET STANDARD), Sept. 1981.

[89] Postel, J. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), Sept. 1981.

[90] Postel, J., and Reynolds, J. File Transfer Protocol. RFC 959 (INTERNET STANDARD), Oct. 1985.

[91] Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., and Raghavan, B. Tcp fast open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (New York, NY, USA, 2011), CoNEXT '11, ACM, pp. 21:1–21:12.

[92] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 266–277.

[93] Raiciu, C., Handley, M., and Wischik, D. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356 (Experimental), Oct. 2011.

[94] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How hard can it be? designing and implementing a deployable multipath TCP. In *NSDI* (2012), vol. 12, pp. 29–29.

[95] Reis, C., et al. Detecting in-flight page changes with web tripwires. In *Symposium on Networked System Design and Implementation (NSDI)* (Apr. 2008).

[96] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G. J., and Lear, E. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), Feb. 1996.

[97] RESCORLA, E., AND MODADUGU, N. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), Apr. 2006.

[98] ROSENBERG, J. UDP and TCP as the new waist of the Internet hourglass, Feb. 2008. Internet-Draft (Work in Progress).

[99] ROSS, K., AND KUROSE, J. *Computer networking : A top-down Approach Featuring the Internet*. Addison Wesley, 2012.

[100] SALTZER, J., REED, D., AND CLARK, D. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS) 2*, 4 (1984), 277–288.

[101] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD), July 2003.

[102] SEMKE, J., MAHDAVI, J., AND MATHIS, M. Automatic tcp buffer tuning. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 1998), SIGCOMM '98, ACM, pp. 315–323.

[103] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making middleboxes someone else's problem: network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 13–24.

[104] SRISURESH, P., FORD, B., SIVAKUMAR, S., AND GUHA, S. NAT Behavioral Requirements for ICMP. RFC 5508 (Best Current Practice), Apr. 2009.

[105] STEWART, R. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007.

[106] STEWART, R., RAMALHO, M., XIE, Q., TUEXEN, M., AND CONRAD, P. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758 (Proposed Standard), May 2004.

[107] STONE, J., AND PARTRIDGE, C. When the CRC and TCP checksum disagree. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2000), SIGCOMM '00, ACM, pp. 309–319.

[108] STRAYER, T., DEMPSEY, B., AND WEAVER, A. *XTP: The Xpress transfer protocol*. Addison-Wesley Publishing Company, 1992.

[109] TOUCH, J. TCP Control Block Interdependence. RFC 2140 (Informational), Apr. 1997.

[110] TUEXEN, M., AND STEWART, R. UDP Encapsulation of SCTP Packets for End-Host to End-Host Communication. Internet draft, draft-ietf-tsvwg-sctp-udp-encaps, work in progress, March 2013.

[111] VASUDEVAN, V., PHANISHAYEE, A., SHAH, H., KREVAT, E., ANDERSEN, D., GANGER, G., GIBSON, G., AND MUELLER, B. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (New York, NY, USA, 2009), SIGCOMM '09, ACM, pp. 303–314.

[112] VELTEN, D., HINDEN, R., AND SAX, J. Reliable Data Protocol. RFC 908 (Experimental), July 1984.

[113] VUTUKURU, M., BALAKRISHNAN, H., AND PAXSON, V. Efficient and Robust TCP Stream Normalization. In *IEEE Symposium on Security and Privacy* (2008), IEEE, pp. 96–110.

[114] W3C. The WebSocket API (draft), 2011. `http://dev.w3.org/html5/websockets/`.

[115] WANG, Z., QIAN, Z., XU, Q., MAO, Z., AND ZHANG, M. An untold story of middleboxes in cellular networks. In *Proceedings of the ACM SIGCOMM 2011 conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 374–385.

[116] WISCHIK, D., HANDLEY, M., AND BAGNULO, M. The resource pooling principle. *SIGCOMM Comput. Commun. Rev. 38*, 5 (Sept. 2008), 47–52.

[117] WISCHIK, D., RAICIU, C., GREENHALGH, A., AND HANDLEY, M. Design, implementation and evaluation of congestion control for Multipath TCP. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 8–8.

[118] ZEC, M., MIKUC, M., AND ZAGAR, M. Estimating the impact of interrupt coalescing delays on steady state TCP throughput. In *SoftCOM* (2002).

[119] ZIMMERMANN, H. OSI reference model–The ISO model of architecture for open systems interconnection. *Communications, IEEE Transactions on 28*, 4 (1980), 425–432.

A Exercises

This appendix contains a few exercises on the evolution of transport protocols and their interactions with middleboxes.

A.1 Transport protocols

1. TCP provides a reliable transport service. Assuming that you control the two endpoints of a connection, how would you modify the TCP protocol to provide an unreliable service ? Explore two variants of such a transport service :

   - an unreliable bytestream where bytes can be corrupted but where there are no losses
   - an unreliable bytestream that prevents data corruption but can deliver holes in the bytestream

2. Same question as above, but for SCTP.

3. TCP provides a connection-oriented bytestream service. How would you modify TCP to support a message-oriented service. Consider two variants of this service :

   - A connection-oriented message-mode service that supports only small messages, i.e. all messages are smaller than one segment.
   - A connection-oriented message-mode service that supports any message length.

4. The large windows extension for TCP defined in [63] uses the `WScale` option to negotiate a scaling factor which is valid for the entire duration of the connection. Propose another method to transport a larger window by using a new type of option inside each segment. What are the advantages/drawbacks of this approach compared to [63] assuming that there are no middleboxes ?

5. One of the issues that limits the extensibility of TCP is the limited amount of space to encode TCP options. This limited space is particularly penalizing in the SYN segment. Explore two possible ways of reducing the consumption of precious TCP option-space

    - Define a new option that packs several related options in a smaller option. For example try to combine SACK with Timestamp and Window scale in a small option for the SYN segment.
    - Define a compression scheme that allows to pack TCP options in fewer bytes. The utilisation of this compression scheme would of course need to be negotiated during the three way handshake.

A.2 Middleboxes

Middleboxes may perform various changes and checks on the packets that they process. Testing real middleboxes can be difficult because it involves installing complex and sometimes costly devices. However, getting an understanding of the interactions between middleboxes and transport protocols can be useful for protocol designers.

A first approach to understand the impact of middleboxes on transport protocols is to emulate the interference caused by middleboxes.This can be performed by using `click` [68] elements that emulate the operation of middleboxes [56] :

- `ChangeSeqElement` changes the sequence number in the TCP header of processed segments to model a firewall that randomises sequence numbers
- `RemoveTCPOptionElement` selectively removes a chosen option from processed TCP segments
- `SegSplitElement` selectively splits a TCP segment in two different segments and copies the options in one or both segments
- `SegCoalElement` selectively coalesces consecutive segments and uses the TCP option from the first/second segment for the coalesced one

Using some of these `click` elements, perform the following tests with one TCP implementation.

1. Using a TCP implementation that supports the timestamp option defined in [63] evaluate the effect of removing this option in the `SYN`, `SYN+ACK` or regular TCP segments with the `RemoveTCPOptionElement` `click` element.
2. Using a TCP implementation that supports the selective acknowledgement option defined in [74] predict the effect randomizing the sequence number in the TCP header without updating anything in this option as done by some firewalls. Use the `ChangeSeqElement` `click` element to experimentally verify your answer. Instead of using random sequence numbers, evaluate the impact of logarithmically increasing/decreasing the sequence numbers (i.e. +10, +100, +1000, +1000, . . .)
3. Recent TCP implementations support the large windows extension defined in [63]. This extension uses the `WScale` option in the `SYN` and `SYN+ACK` segments. Evaluate the impact of removing this option in one of these segments with the `RemoveTCPOptionElement` element. For the experiments, try to force the utilisation of a large receive window by configuring your TCP stack.
4. Some middleboxes split or coalesce segments. Considering Multipath TCP, discuss the impact of splitting and coalescing segments on the correct operation of the protocol. Use the Multipath TCP implementation in the Linux kernel and the `SegCoalElement` and `SegSplitElement` `click` elements to experimentally verify your answer.

5. The extensibility of SCTP depends on the utilisation of chunks. Consider an SCTP-aware middlebox that recognizes the standard SCTP chunks but drops the new ones. Consider for example the partial-reliability extension defined in [106]. Develop a `click` element that allows to selectively remove a chunk from processed segments and evaluate experimentally its impact on SCTP.

Another way to evaluate middleboxes is to try to infer their presence in a network by sending probe packets. This is the approach used by Michio Honda and his colleagues in [58]. However, the TCPExposure software requires the utilisation of a special server and thus only allows to probe the path towards this particular server. An alternative is to use `tracebox` [33]. `tracebox` is an extension to the popular `traceroute` tool that allows to detect middleboxes on (almost) any path. `tracebox` sends TCP and UDP segments inside IP packets that have different Time-To-Live values like `traceroute`. When an IPv4 router receives an IPv4 packet whose TTL is going to expire, it returns an ICMPv4 *Time Exceeded* packet that contains the offending packet. Older routers return in the ICMP the IP header of the original packet and the first 64 bits of the payload of this packet. When the packet contains a TCP segment, these first 64 bits correspond to the source and destination ports and the sequence number. However, recent measurements show that a large fraction of IP routers in the Internet, notably in the core, comply with [10] and thus return the complete original packet. `tracebox` compares the packet returned inside the ICMP message with the original one to detect any modification performed by middleboxes. All the packets sent and received by `tracebox` are recorded as a libpcap file that can be easily processed by using `tcpdump` or `wireshark`.

1. Use `tracebox` to detect whether the TCP sequence numbers of the segments that your host sends are modified by intermediate firewalls or proxies.

2. Use `tracebox` behind a Network Address Translator to see whether `tracebox` is able to detect the modifications performed by the NAT. Try with TCP, UDP and regular IP packets to see whether the results vary with the protocol. Analyse the collected packet traces.

3. Some firewalls and middleboxes change the `MSS` option in the `SYN` segments that they process. Can you explain a possible reason for this change ? Use `tracebox` to verify whether there is a middlebox that performs this change inside your network.

4. Use `tracebox` to detect whether the middleboxes that are deployed in your network allow new TCP options, such as the ones used by Multipath TCP, to pass through.

5. Extend `tracebox` so that it supports the transmission of SCTP segments containing various types of chunks.

A.3 Multipath TCP

Although Multipath TCP is a relatively young extension to TCP, it is already possible to perform interesting experiments and simulations with it. The following resources can be useful to experiment with Multipath TCP :

- `http://www.multipath-tcp.org` provides an implementation of Multipath TCP in the Linux kernel with complete source code and binary packages. This implementation covers most of [43] and supports the coupled congestion control [93] and OLIA [66]. Mininet and netkit images containing the Multipath TCP kernel are available from the above website.

- http://caia.swin.edu.au/urp/newtcp/mptcp/ provides a kernel patch that enables Multipath TCP in the FreeBSD-10.x kernel. This implementation only supports a subset of [43]
- The ns-3 network simulator[10] contains two forms of support for Multipath TCP. The first one is by using a Multipath TCP model[11]. The second is by executing a modified Linux kernel inside ns-3 by using Direct Code Execution[12].

Most of the exercises below can be performed by using one of the above mentioned simulators or implementation.

1. Several congestion control schemes have been proposed for Multipath TCP and some of them have been implemented. Compare the performance of the congestion control algorithms that your implementation supports.
2. The Multipath TCP congestion control scheme was designed to move traffic away from congested paths. TCP detects congestion through losses. Devise an experiment using one of the above mentioned simulators/implementation to analyse the performance of Multipath TCP when losses occur.
3. The non-standard TCP_INFO socket option[84] in the Linux kernel allows to collect information about any active TCP connection. Develop an application that uses TCP_INFO to study the evolution of the Multipath TCP congestion windows.
4. Using the Multipath TCP Mininet or netkit image, experiment with Multipath TCP's fallback mechanism by using ftp to transfer files through a NAT that includes an application level gateway. Collect the packet trace and verify that the fallback works correctly.

[10] See http://www.nsnam.org/.
[11] See https://code.google.com/p/mptcp-ns3/
[12] See http://www.nsnam.org/projects/direct-code-execution/