A New Backward-Compatible Web Transport

Nabin Tiwari ‘11

A thesis submitted to the Faculty of Franklin & Marshall College in partial fulfillment of the requirements for Honors in Computer Science

May 2011

Honors Advisor: Prof. Janardhan Iyengar
Honors Committee:
  Prof. Jing Hu, Computer Science
  Prof. Erik Talvitie, Computer Science
  Prof. Iwan Praton, Mathematics (External)
Contents

Abstract ................................................................................................................................. 4
Section 1: Understanding Internet Communication ......................................................... 4
  1.1 An Internet Communication Model ........................................................................... 4
  1.2 The Internet Service Model ..................................................................................... 5
  1.3 The Modern Internet ................................................................................................. 6
    1.3.1 Middleboxes in the Internet ................................................................................. 6
    1.3.2 Implications: Hurdles in Deploying New Transport Services ......................... 6
Section 2: The Web and TCP ............................................................................................ 7
  2.1 Non-Persistent and Persistent Connections ............................................................ 8
  2.2 Problems with Multiple TCP Connections .............................................................. 8
  3.1 uTCP: Breaking Out of the Transport Logjam ......................................................... 9
    3.1.1 Modifications to the TCP Stack and the TCP API: ........................................... 9
    3.1.2 Limitations of uTCP ........................................................................................ 10
  3.2 uCOBS: Building Unordered Messages .................................................................. 11
    3.2.1 COBS (Consistent-Overhead Byte Stuffing) .................................................... 11
  3.3 uHTTP: Using an Unordered Transport for the Web ............................................... 12
Section 4: Implementation of Our New Transport .......................................................... 13
  4.1 uTCP Implementation .............................................................................................. 13
  4.2 uCOBS Implementation .......................................................................................... 13
  4.3 uHTTP Implementation ........................................................................................... 14
  4.4 Implementation Challenges .................................................................................... 15
Section 5: Results ............................................................................................................. 15
  5.1 Basic Functionality of uTCP .................................................................................... 15
  5.2 Bandwidth and CPU Costs ...................................................................................... 17
Section 5: Related Work .................................................................................................. 18
Section 6: Conclusions and Future Work ...................................................................... 18
Section 7: References ..................................................................................................... 19
Abstract
The widely-used TCP protocol for Internet communication provides a reliable and ordered transfer of a stream of bytes between two communicating devices. A significant drawback of TCP is that it does not support parallelism within its transfer, making it less suitable as a transport protocol for transferring Web pages. To reduce this problem, web browsers commonly use multiple TCP connections to stream data in parallel, which introduces its own problems and does not eliminate TCP’s fundamental limitations. Middleboxes—devices that were not part of the original Internet architecture but are now pervasive—such as NATs, firewalls and PEPs (performance enhancing proxies) are designed and optimized for TCP only. These devices violate the Internet’s end-to-end architecture and prevent any new transport protocol from being deployed over the network. In this project, we develop a new web transfer protocol that uses unordered-TCP (uTCP), our extension of TCP that allows unordered delivery of messages without modifying the TCP protocol, to enable parallelism within a TCP stream without affecting its ability to traverse middleboxes. We discuss the design and the implementation of the new web transport protocol (uHTTP), uTCP and the user-space library uCOBS. We also present some results that show the benefits of uTCP.

Section 1: Understanding Internet Communication

The Internet is a massive global network of communication links and packet switches that enables digital communication between millions of devices such as computers, cell phones, and game consoles. All of these devices, which are the ultimate generators and consumers of data on the network, are called end-hosts or end-systems. Any end-to-end communication happens in the form of packets that traverse through packet-switches---devices such as routers and link layer switches that forward packets to appropriate destinations and tie the network together.

1.1 An Internet Communication Model

Network communication can be modeled as a hierarchy of layers where each layer performs a well-defined function and provides services to the layer above it. In such model, layer n provides well-defined services to layer n+1 but hides the details of how the services are actually implemented. When two hosts communicate with each other, information first originates at the highest level and passes down the layers until it reaches the lowest layer. Each layer encapsulates information by adding a layer-specific header to it. The lowest layer is responsible for the physical transfer of bits. Once the encapsulated information reaches the destination host, it travels up the stack, with headers being removed at corresponding layers on the way up the stack, until the information reaches the highest layer.

In a layered model, the hardware or software entities that are on the same layer are known as peers. Peers at layer n of one machine communicate with peers at layer n of another machine in the network using a specific protocol for the communication. A protocol is a formal specification of the precise format and the order of messages exchanged between peers. A protocol also defines how a communicating entity responds to any messages or events that it receives. Even though data passes vertically up or down the stack, communication occurs horizontally across peers in the same layer.

The TCP/IP model is widely used to model Internet communication. The five different layers of the model are briefly described below.

Layer 5, The Application layer: The application layer is the uppermost layer in the stack. This layer consists of network applications and their corresponding protocols. Some of the protocols used in this layer are HTTP (for web applications), FTP (for file transfer) and SMTP (for email).
Layer 4, The Transport Layer: This layer provides end-to-end data transfer service between communicating applications. The most common transport layer protocols are TCP (Transport Control Protocol) and UDP (User Datagram Protocol). TCP provides ordered, reliable data transfer. The reliability property of TCP ensures that any data sent by one end reaches the other end. UDP provides unreliable and unordered datagram service. UDP only delivers the messages that made it through the network without corruption. Unlike TCP, UDP does not try to recover any lost or corrupt messages.

Layer 3, The Network layer: This layer tries to find the best path through the network between the communicating end-hosts. It also forwards data packets along these paths. The Internet’s network layer only provides a best-effort service and does not try to recover from data loss or corruption.

Layer 2, The Link layer: The network route between two end-hosts goes through nodes such as routers and switches; the link layer is responsible for the transfer of packets between any two nodes that lie along a route. An example link layer protocol is Ethernet.

Layer 1, The physical layer: This layer is concerned with the physical transfer of actual data bits between two nodes in the network; examples are copper wire and optical fiber.

1.2 The Internet Service Model

The Internet, as a single network, is unified by the use of one network layer protocol on the data path between two end-hosts: the Internet Protocol (IP). IP provides a best-effort datagram service to the Transport Layer. Even though IP tries its best to deliver all datagrams (or packets), it does not provide reliability. It does not guarantee that a datagram sent by a source host reaches the destination host. Nor does it guarantee that datagrams arrive at the receiving host in the same order as sent by the sending host. This is reasonable because IP is designed to work on top of all kinds of link layer devices and to support applications that have different reliability needs. It is the Transport Layer’s job to provide a reliable service by building on top of the Network layer. The Internet Protocol uses IP addresses, which uniquely identify hosts connected to the Internet, to perform routing and forwarding.

There are two main protocols at the Transport layer - Transport Control Protocol (TCP) and User Datagram Protocol (UDP). These two protocols are currently the workhorses of the Internet and are used by all major applications on the Internet. TCP provides a ordered reliable transfer of bytestream. It has various mechanisms for providing reliability, such as timeouts and retransmissions that try to recover any lost packet. It also provides flow control and congestion control. The flow control in TCP ensures that a sender sends data at a rate at which a client can reliably process the data. The flow control mechanism prevents a sender from overwhelming a device with slow processing speed or low memory capacity. The congestion control mechanism in TCP ensures that end-hosts do not send data at a rate that overwhelms the routers and packet switches in the network. TCP is currently the most widely used protocol. Many applications such as the web, file transfer applications, email, and instant messaging use TCP.

UDP provides a very simple unordered datagram service. It does not try to recover lost datagrams and does not reorder datagrams. It, however, has a checksum mechanism that ensures that any datagram that is received is error-free. UDP is used by applications that do not need reliability and/or by applications that wish to create their own mechanisms for reliability, congestion control and/or flow control. Applications such as video conferencing (skype), media streaming and multiplayer online games commonly use UDP.
1.3 The Modern Internet

1.3.1 Middleboxes in the Internet

The Internet has changed as it tried to adapt to the various needs of its users. The past 10 years have seen a mushrooming of devices in the network that are neither routers nor end-systems and are referred to as middleboxes. The most common middleboxes are Network Address Translaters (NATs), firewalls and Performance Enhancing Proxies (PEPs). NATs, commonly known as home routers, are cheap and are widely used middleboxes that allow many home devices such as game consoles, laptops and desktops to share a single connection to access the Internet. The devices behind the NAT get private IP addresses but they share the same public IP address. NATs change the IP header in all packets that go in and out of a private network to allow all of the devices in the private network to connect to the Internet, and more importantly, they also change the transport header. Firewalls are software or hardware implementations that inspect all packets that go in and out of a network to prevent intrusion into the network. They inspect TCP and UDP port numbers in packets to detect and possibly drop packets from unauthorized or blocked applications. Firewalls are very popular and are widely used in corporate networks and in many modern operating systems to enhance security. PEPs are devices that increase the performance of network connection over non-traditional links such as the satellite links. These devices only work with TCP and UDP. Even though UDP gets through many middleboxes, some middleboxes such as firewalls also block UDP because many rogue applications like viruses and Trojan horses use UDP as their Transport protocol. The Internet architecture was not designed with the middleboxes in mind. However, these devices have become pervasive and are slowing down the evolution of the Internet.

1.3.2 Implications: Hurdles in Deploying New Transport Services

While TCP seems to be the most accepted transport among all middleboxes, it is not without its limitations. TCP provides strict reliability and in-order delivery to the application. With TCP, a receiver receives data in the same order as sent by the sender, and the sender ensures that data is reliably received. Many applications such as the web, video-conferencing, and remote desktop applications do not require this strict in-order delivery or the strict reliability service of TCP. These applications can face severe delays in lossy networks due to the feature. If a packet gets dropped in the network, all subsequent packets are queued by TCP until the dropped packet is retransmitted and received. Even if an application can immediately use the data in subsequent packets, it will have to wait for the lost packet to be received---this problem is known as head-of-line blocking. This problem is particularly exacerbated in lossy networks, such as is often the case in cellular access networks and with satellite networks. Latency-sensitive applications can simply use UDP; however many choose to use TCP so that they can readily work on all networks and through all middleboxes. For instance, Skype, the popular video-conferencing application, falls back to TCP whenever communication over UDP fails [1].

New Transport Protocols such as SCTP (Stream Control Transport Protocol), and SST (Structured Stream Transport) have been designed to provide much richer services, such as multihoming, which enables an application to use multiple interfaces (IP addresses) simultaneously to connect to the Internet; and multistreaming, which allows applications to interleave many independent streams and/or objects over a single end-to-end transport connection and avoid the head-of-line blocking problem. Even though several applications, such as the web, can benefit from new transport protocols, any new transport encounters significant deployment hurdles, since middleboxes are now largely designed and optimized for TCP only. Deployment of any new transport protocol is almost impossible because of the pervasiveness of middleboxes that generally violate the end-to-end principle of network communication [11, 12], and almost always fail to work for new transport protocols [7]. Simply put, we are stuck with TCP.
To break out of the logjam of the need for new transports and the inability to deploy any new ones, we now present uTCP, a TCP modification to enable out-of-order delivery through endhost modifications, without changing how TCP looks on the wire. uTCP appears as TCP to middleboxes in the network, thus managing to get through the legacy Internet, and allowing more services, such as better services for the web, to be built on top.

In Section 2, we describe why TCP is not suitable as a web transport. In Section 3, we first describe uTCP and then our unordered message service with uCOBS atop uTCP. We then present an unordered web protocol with uHTTP as a simple extension to the standard HTTP. In Section 4, we describe our implementation of uTCP, uCOBS, and uHTTP, followed by experimental evaluation of uTCP and uCOBS in Section 5. Section 6 concludes the thesis with thoughts on future work.

It is important to note that while we do not resolve all of the web’s transport issues in this work, new transport services, such as interleaving of objects, unordered object delivery, and object prioritization all require out-of-order delivery. uTCP provides this substrate on top of which we can build the rest of the services needed for a more sophisticated web transport.

Section 2: The Web and TCP

HyperText Transfer Protocol (HTTP) is an application-layer protocol that underlies the World Wide Web. HTTP is based on client-server architecture - the client sends a request for an object (usually a file) to the server, which in turn, sends the object back to the client. The client and the server communicate with each other via HTTP messages which are defined by the HTTP protocol. HTTP runs on top of TCP, so the protocol only works for strictly ordered delivery. Currently, there is no feature in HTTP that lets us perform out of order delivery; however, we examine how a slight modification to HTTP allows us to perform out of order delivery.

A web page is a collection of several independent objects such as the HTML files, CSS stylesheets, JavaScript files, images, videos and applets. After a browser receives the index page and the CSS stylesheets, the browser can immediately start displaying the web page. A web client first requests the index.html file. The file contains links to embedded objects. These links are also known as URLs (Universal Resource Locators). After the client receives the index page, it sends HTTP requests for the embedded objects. The browser can then receive the remaining objects in any order, since these objects are independent of each other. As soon as the client receives the index page and the CSS files, it can start displaying the web page. The strict in-order delivery of TCP can cause unnecessary head-of-line blocking, since these independent objects are often sent over the same TCP connection. For example, if a TCP segment containing a piece of an image gets dropped in the network, then TCP queues any subsequent segments in its out-of-order queue until it recovers the lost packet. Even if the new segments contain objects that the browser can immediately display to the user, the browser cannot do so because TCP is holding these objects in its delivery queue.

Browsers try to mitigate the head-of-line blocking problem by establishing multiple connections to download web objects in parallel. Even if a packet in one of the connection gets lost, the browsers can still use the data coming from the other connections. However, there are several disadvantages to this method which we now describe in section 2.2.
2.1 Non-Persistent and Persistent Connections

In a non-persistent HTTP connection, a separate TCP connection is created for each object. This was the default mode in the older HTTP 1.0. On the other hand, in a persistent HTTP connection, the same TCP connection is used to transfer all objects. Persistent connections are the default mode in HTTP 1.1 and can be established by adding the option “Connection: Keep-Alive” to the HTTP header. Since it requires one round trip time (RTT) between the client and the server to establish a TCP connection and at least one more RTT to request and receive an object, the total time required to transfer n objects in a non-persistent connection is around 2n X RTT. On the other hand, a persistent connection only needs to establish a TCP connection once, so it downloads n objects in about (n+1) X RTT. Most browsers and web servers today use several persistent connections, which reduces head-of-line blocking and seeks to limit the problems with multiple connections. We now describe problems associated with establishing multiple connections to download a webpage.

2.2 Problems with Multiple TCP Connections

There are several problems associated with the practice of using multiple TCP connections to download webpages [10]. Some of them are briefly described below.

1. A server needs to maintain state information for every TCP connection. Thus, multiple TCP connections increase the load on a server.

2. TCP’s loss detection and recovery mechanisms work well over connections that last for a longer period of time. When data is downloaded over multiple connections, each connection lasts for a comparatively shorter period of time. Therefore, the use of multiple TCP connections does not always improve performance over lossy networks. For instance, TCP has a loss recovery mechanism called fast-retransmission which kicks in when the sender receives three duplicate acknowledgements. Since there are fewer packets flowing in the network in a shorter connection and different connections do not share information with each other, the chances of a server receiving three duplicate acknowledgements is low. This decreases the chances of loss recovery via fast-retransmission.

3. TCP uses three-way handshake to establish a connection. First a client sends SYN packet (TCP packet with the Synchronize flag turned on). The server responds with a SYN-ACK packet (TCP packet with Synchronize flag and Acknowledgement flag turned on). The client finally responds with an ACK packet. TCP waits for a much longer time if there is a SYN packet loss in the network before retransmitting the SYN packet. In a lossy network, the chances of losing the SYN packet gets higher with the increase in number of connections because each connection involves the three-way handshake. So, multiple connections can increase the total download time in lossy networks.

4. As mentioned above, browsers use multiple TCP connections to avoid the head-of-line blocking problem. However, using multiple connections actually has the side effect of providing more bandwidth to an application at the cost of lower bandwidth to some other users or applications. So the use of multiple connections creates fairness problems. The HTTP 1.1 specification recommends that browsers not open more than two persistent connections to a server. However, some browsers started using multiple persistent connections and soon other browsers followed to avoid being penalized by the fairness issue.

In summary, instead of actually solving the head-of-line blocking problem, the practice of establishing multiple TCP connections leads to a cascade of other problems. As we discussed earlier, attempts to build new transports that solve this problem effectively do not work because of deployment issues with new transports. In the following section, we present uTCP and describe how it minimizes the head-of-line blocking problem without introducing the problems associated with multiple TCP connections.
Section 3: Designing a Backward-compatible Web Transport

3.1 uTCP: Breaking Out of the Transport Logjam

We introduce uTCP, a modification of TCP to provide out-of-order delivery at a receiver. The most important goal that drives uTCP’s design is network compatibility—we want uTCP to get through all middleboxes in the network that allow TCP through. The only way to do this is to make uTCP look exactly like TCP on the wire. This design constraint forces us to avoid any changes to the actual TCP protocol, and uTCP, therefore, modifies how TCP works at the end-hosts without changing how TCP looks on the wire and to middleboxes in the network.

A second design goal is to minimize any changes to the TCP stack inside the Operating System kernel, to reduce hurdles to deployment. uTCP only provides out-of-order delivery of contiguous “chunks” of data. We build message encoding, decoding and framing features in a userspace library, as described later in this section. To be able to perform these functions in userspace, some information is required to gather these chunks of data and to be able to extract out-of-order records from these chunks. The TCP sequence number is therefore also delivered as metadata with any data that uTCP delivers up to userspace.

3.1.1 Modifications to the TCP Stack and the TCP API:

We now describe the modifications we made to TCP stack in the Linux 2.6.34 kernel. Similar modifications can be made to the TCP stack in other Operating System kernels as well.

We added a SO_UNORDERED socket option to the Linux kernel. When this option is turned ON for an end of a TCP connection, the end TCP stack behaves as uTCP, delivering chunks of bytes out-of-order, as they are received by TCP.

An application that wishes to use uTCP first goes through the same process as involved in starting a TCP connection. The application first creates a stream socket and uses connect() or accept() to establish a TCP connection. As soon as the application is ready to receive data out-of-order from the connection, it invokes setsockopt() and enables the SO_UNORDERED socket option.

The kernel code that serves the TCP read() call behaves differently when the SO_UNORDERED socket option is enabled. When the option has not been enabled as in normal TCP, the kernel queues any out-of-order segments in the out-of-order queue but does not return any data to the user until holes in the queue are filled. Figure 1 shows how TCP and uTCP behavior differ after a data packet loss. Figure 1a shows that TCP receives a segment of 100 bytes starting at sequence number 101 and ending at sequence number 200, in-order. The second diagram in Figure 1a shows the arrival of an out-of-order segment of 100 bytes starting at sequence number 301 and ending at sequence number 400. This out-of-order segment is a result of a packet drop that affected the packet containing TCP segment 201-300. TCP puts the out-of-order segment 301-400 in its out-of-order queue and waits for the arrival of 201-300 segment. The third diagram shows the arrival of the 201-300 segment. This time TCP delivers both segments 201-300 and 301-400 to the receiving application. Figure 1b shows that unlike TCP, uTCP delivers the out-of-order segment 301-400 immediately upon its arrival.
Whenever a host tries sending a large chunk of data over TCP, TCP first breaks the data into manageable segments before transmitting them over the network. This means that a large file, such as an image file, is transferred as many TCP segments. Any receiving application must put the segments together to create a complete file. In the case of TCP, since an application receives all the segments in order, it just copies the data in the segments sequentially to a file to recreate the original file. However, in the case of uTCP, this task gets more challenging because an application can receive the segments in any order. To recreate the original file, the application first needs to put the segments in order. For this reason, uTCP appends TCP sequence number to any data segment that it delivers to the application. Along with the sequence number it also appends a flag byte that indicates the application if the segment is delivered out-of-order. When uTCP delivers a segment to the application, the first 5 bytes contain metadata (4 bytes for TCP sequence number and 1 byte for flag) that allows the application to put the segments in order and extract any message. We note that this modification is an API-only and endhost-only modification, and does not change the wire-format of the protocol.

3.1.2 Limitations of uTCP

In the example in Figure 1, even after uTCP delivers the out-of-order 301-400 segment, the segment is still put in the out-of-order queue in the kernel. This is done to minimize any changes to the kernel code. When uTCP receives the lost segment 201-300, it delivers everything from 201-400. So in this case, the user application receives the 301-400 segment twice. Even though this problem of duplicate data delivery can be avoided by adding some code in the kernel, we decided not to do so to avoid adding complexity to the kernel code. The uCOBS userspace library, as we describe next, has mechanisms to make sure that an application does not receive duplicates.
3.2 uCOBS: Building Unordered Messages

uTCP provides chunks of bytes up from the TCP stack; uCOBS is a userspace library that uses the uTCP kernel modification and API to provide out-of-order datagram (message) delivery service to an application. To provide message-oriented delivery, uCOBS needs a mechanism to extract messages from the data chunks it receives from the kernel. To provide out-of-order delivery, uCOBS uses markers to mark the start and the end of a message.

3.2.1 COBS (Consistent-Overhead Byte Stuffing)

When markers are used to demarcate a message, we need to eliminate the marker byte from the actual message so that any occurrence of the marker byte in the data stream are not confused with the inserted message markers. To eliminate any occurrences of the marker byte from a message, we use Consistent Overhead Byte Stuffing (COBS), a binary encoding which eliminates the occurrence of a given byte from a message with a fixed maximum size overhead [4]. Since COBS allows us to remove any byte value, we use the zero byte as our marker. All following explanations assume that zero byte is used as the marker byte.

COBS works on byte level, which means that it considers any message to be composed of 256 different byte values starting from the zero byte (00000000 or 0x00) and ending at the 255th byte (11111111 or 0xFF). COBS first scans the message to find the index of the first zero byte (Indexing starts at 1 – the first byte has index 1). It then removes the zero byte from the message and instead appends the index value to the start of the message. For instance, if the zero occurs at 10th position, it removes the zero and appends the tenth byte (0xA) to the message. However, since there are only 256 bytes, if the zero occurs beyond the 255th position, COBS cannot use any byte to indicate the position of that zero. This problem is overcome by using the 0xFF (the 255th byte) to denote that there is no zero byte in the first 254 data bytes.

After COBS removes the first zero byte or after it uses 0xFF to indicate the absence of zero byte, it starts counting the next unprocessed byte starting from index 1 and repeats the above process of replacing zero bytes. COBS assumes that there is an implicit zero byte at the end of the message to properly encode the last block. The following example illustrates COBS encoding.

Original Message

| 67 | 98 | 99 | 102 | 107 | 0 | 5 | 0 | 0 | 67 | 68 |

Cobs-encoded message

| 6 | 67 | 98 | 99 | 102 | 107 | 2 | 5 | 1 | 3 | 67 | 68 |

If the the distance between the zero bytes in the original message is always less than 255, then uCOBS incurs an overhead of one byte only as shown in the illustration shown. The overhead in the pathological case when there are no zero bytes in the original message is 1/255 = 0.4%. Thus,

---

1 Applications like HTTP and SIP that run on top TCP traditionally use Type-Length-Value (TLV) encodings for marking message boundaries. These encodings usually assume in-order delivery, and need to process the TLV header of the first message before they can parse the TLV header of subsequent messages.
COBS incurs a maximum overhead of 1 byte for messages smaller than 255 bytes, and for larger messages the maximum overhead is 1 byte plus 0.4% of the total size. For a 1000-byte message, this leads to a maximum overhead of 5 bytes.

After eliminating all the zero bytes from a message, a COBS encoder inserts a zero marker-byte at the beginning and at the end of the message, effectively demarcating the message. At the receiving side, a COBS decoder can then search for zero bytes in the segments received from uTCP to extract a message, and decode the COBS-encoded message using the reverse of the COBS-encoding algorithm. uCOBS provides two API methods cobs_sendmsg() and cobs_recvmsg(). These methods let an application send and receive COBS-encoded messages.

### 3.3 uHTTP: Using an Unordered Transport for the Web

Now that we have achieved unordered message delivery with uTCP and uCOBS, we introduce uHTTP, a modified version of HTTP that supports unordered processing of HTTP requests and responses. uHTTP uses uCOBS over a single uTCP connection to transfer web objects and minimizes the head-of-line blocking problem **within a single TCP connection**, thus avoiding the problems associated with creating multiple TCP connections.

The design of HTTP assumes that TCP is the underlying transport and as a result, a client expects the server to respond to its HTTP requests in the order that the requests are sent by the client. Since the goal of uHTTP is to support unordered delivery of web objects, we now need to extend HTTP to support an unordered transport underneath. HTTP provides a clean method for us to add the necessary information for supporting an unordered transport: HTTP requests and responses have extensible headers with as many headers as the client/server want to send each other, and we simply add our own to this list. We introduce a new HTTP header field called “Object-ID”. Whenever a client sends a request to a server, it adds an opaque Object-ID field along with a unique identifier for the object. Then server simply echoes back this identifier in its response. This attached Object-ID field allows a client to map any response to its corresponding request.

We want the modified HTTP to be backward-compatible to HTTP. This means that a uHTTP client should fall back to normal HTTP if the server does not support uHTTP. To maintain this behavior, the client and the server first negotiate the use of uHTTP over plain HTTP; if this negotiation fails, the communication must fall back to plain HTTP. We use the same Object-ID HTTP header field for our negotiation. The client initially uses plain HTTP over TCP to send an HTTP request, and in this request, adds the Object-ID header field. If the server supports uHTTP and sees the Object-ID field, it assumes that the client supports uHTTP and starts using uHTTP from that point onwards. In its response to the client, the server also adds the Object-ID header field, thus indicating its own support of uHTTP to the client. When the response is received by the client, it too switches to using uHTTP.

The client and server can negotiate the type of encoding used for uHTTP using pre-existing header fields - “Accept-Encoding” and “Content-Encoding”. In our case, we specify COBS as the encoding of the response data. However, the client and the server can negotiate the use of other encodings in the future as well. When a client sends a request to the server, it adds the “Accept-Encoding: Cobs” to the request header. If the server supports uCOBS, then it adds “Content-Encoding: COBS” field to its COBS-encoded uHTTP response. The server uses httpcobs_sendmsg(), a slightly different version of cobs_sendmsg() described earlier, to send a message. To be consistent with the HTTP specification, which specifies that the content-encoding field should be used to describe the encoding of the message body only, and since HTTP headers do not contain zero bytes, we do not encode the HTTP header, but only the message body. Thus, httpcobs_sendmsg() does not COBS-encode the header, but only encodes the data: it COBS-encodes the message body, concatenates the un-encoded header with the encoded message body, and adds start and
end zero-markers to the resulting object. The following diagram illustrates how httpcobs_sendmsg() forms a message.

| Start zero marker | HTTP Header | COBS-encoded message body | End zero marker |

**My contribution:**
I was personally involved on the design of uTCP, uCOBS and uHTTP. The uTCP design was a collaborative work involving me, Janardhan Iyengar (assistant professor, F&M), Jeff Wise (F&M student) and Syed Obaid Amin (Post-doc researcher at F&M). The uCOBS API design involved me, Jeff Wise and Janardhan Iyengar. uHTTP design was done in collaboration with Janardhan Iyengar and Bryan Ford (assistant professor, Yale University).

## Section 4: Implementation of Our New Transport

### 4.1 uTCP Implementation

Our uTCP implementation adds about 240 lines and modifies about 50 lines of code in the Linux kernel implementation of TCP. Adding the new TCP_UNORDERED socket option in the kernel involves adding code to deliver data from TCP’s out-of-order queue when there is no data in the in-order queue. Our code also modifies the data returned by read() call by appending 5 bytes of metadata to any data that is delivered from the kernel to an application. The first byte is used for 8 flag bits, of which one flag bit is used to indicate to the application if the data is from in-order queue or the out-of-order queue. The remaining four bytes are used to deliver TCP sequence number. Our modifications involves only the read() call for TCP over IPv4. However, the same changes can be easily made to the IPv6 code as well.

Most of the changes occur in the tcp_recvmsg() method in linux/net/ipv4/tcp.c file. This method first checks to see if the socket option TCP_UNORDERED is enabled. If the option is enabled, and there is no data in the TCP in-order queue, then it calls ofo_unordered_delivery() method that we added to the kernel, which walks through the out-of-order queue and delivers the first contiguous data chunk in the out-of-order queue.

### 4.2 uCOBS Implementation

We implement uCOBS as a user-space library in C. It contains approximately 700 lines of C code and provides applications with two functions cobs_recvmsg() and cobs_sendmsg(). The cobs_sendmsg() function first COBS-encodes the message given by a sending application before delivering it to the kernel for transmission. The cobs_recvmsg() function allows an application at the receiving end to receive out-of-order, but complete, messages. We now briefly describe how the uCOBS library provides out-of-order message delivery service.

From this point on, we use the term *byteblock* to refer to any block of data, excluding any metadata (sequence number and the flag byte), that uTCP delivers up to uCOBS. uTCP guarantees that the data in a byteblock is contiguous. In other words, the data within a byteblock appears in proper sequence, with no holes. As mentioned previously, a message can be contained in more than one byteblock. This immediately implies that uCOBS needs a mechanism to arrange the byteblocks according to their starting sequence numbers. uCOBS uses a sorted doubly-linked list for this purpose. A node of the doubly-linked list contains a byteblock, the initial sequence number and the size of the byteblock, a flag to indicate if the byteblock has already been delivered to the receiving
application and a flag to indicate if the byteblock is contiguous to byteblock in the next node. The struct for the byteblock node is given below:

```c
struct byteblock {
    struct byteblock* next;
    struct byteblock* prev;
    uint32_t seq;
    unsigned char* data;
    int size;
    int delivered;
    int contiguous_next;
};
```

The linked list is sorted according to the sequence number of the first byte in the byteblocks. As mentioned in the previous sections, since uTCP can deliver duplicate data to uCOBS, it needs a robust mechanism to detect duplicate data and avoid delivering duplicate messages to the application. As soon as uCOBS receives a byteblock from the kernel, it first checks to see if the linked list already contains some of the data in the new byteblock. If that is the case, uCOBS creates one or more new byteblocks, copies any new data from the original byteblock to the new byteblocks and discards the original byteblock.

uCOBS maintains a cumulative sequence number, which is the sequence number of the last byte received in order. This sequence number is necessary for uCOBS to determine if it can discard any nodes after extracting complete messages from them. If uCOBS extracts and delivers messages from a byteblock which is not within the cumulative sequence number, then it needs to store the information about the byteblock to avoid delivering the same message to the application again. Since uCOBS scans for messages in the sorted list starting from the first byteblock in the list, if the delivered message is within the cumulative sequence number, then uCOBS can eliminate all the list nodes that come before the delivered message, in sequence number. To maintain such cumulative sequence number uCOBS needs a signal from the kernel that if a byteblock is from out-of-order queue or the ordered queue in uTCP. So, along with the initial sequence number, uTCP sends a flag byte to uCOBS. One bit of flag byte is used to indicate if the segment is from out-of-order queue or from ordered queue.

### 4.3 uHTTP Implementation

uHTTP implementation is currently incomplete. The challenging part of the implementation is expected to be the smooth transition to uHTTP after negotiation. Before the client receives a response from the server, it does not know whether or not the server supports uHTTP. One possible way in which a client can transition to uHTTP is to use recv()call with the MSG_PEEK flag set to 1, to determine if the first byte it received from the server is a zero byte. The occurrence of such zero byte implicitly implies that the server is responding with the Object-Id field. A client can further verify the occurrence of Object-Id field by looking at the uHTTP header. The MSG_PEEK flag allows an application to receive data from the TCP queue without removing it from the TCP buffer. If the first byte is a zero byte, then the client enables the TCP_UNORDERED socket option and calls the function httpcobs_recvmsg(), a slightly modified version of cobs_recvmsg() to account for the unencoded HTTP header, to receive the HTTP message. Since the zero byte was not removed from the TCP buffer by the earlier recv()call (since MSG_PEEK was set), httpcobs_recvmsg() receives the starting zero marker, and can decode the message. If, on the other hand, the first byte is not a zero byte, then the client keeps on using regular HTTP over TCP but still adds the "Accept-Encoding: COBS/1.0" header to any HTTP request. The server can switch to using uHTTP using a similar method.
4.4 Implementation Challenges

The majority of the work in the project went into implementing uTCP and uCOBS. uTCP implementation took about a month to complete a first cut. Our uCOBS implementation took about two months to complete. Several additional months were needed to debug both implementations. Implementing uTCP was the most challenging part of the project. It required us to modify the largely undocumented Linux kernel TCP code.

The steps involved in writing, modifying and debugging kernel code are different from the steps involved in working with user-space code. Compiling kernel code takes a very long time (from 10 minutes to 30 minutes) and after every small change or bug fix, the operating system needs a clean restart for the new code to take effect. On average, we had to wait for half an hour to see the results of our changes. Even to make very simple modifications to the kernel code, we had to understand the functions of various variables and data structures used in the code. This was tough because of lack of a comprehensive documentation. To debug kernel code, we used printk() (kernel equivalent of printf). The messages printed go to a special buffer file (the kernel ring buffer) which can be viewed using the command dmesg. This made debugging hard because we had to repeatedly use dmesg to check for asynchronous kernel messages. Overall, it was a different experience working on the Kernel code, and I learnt new aspects of software engineering.

My Contribution:
I and Jeff Wise led the uTCP implementation and later Syed Obaid Amin took over the debugging work. I and Jeff started the uCOBS library implementation, but I wrote most of the functions and completed the implementation. I have been debugging and optimizing the library since fall 2010. I am currently working on the uHTTP implementation.

Section 5: Results

In this section we describe the results of the experiments performed to evaluate the performance of uTCP, uCOBS and uHTTP. All these experiments were conducted on machines running Linux ver 2.6.34 and dummynet was used to emulate various network conditions [2].

These experiments were designed by all members of the Tng group, and mostly run by Syed Obaid Amin and Micheal Fitz Nowlan. I helped Obaid run the experiments involving the basic functionality of uTCP.

5.1 Basic Functionality of uTCP

Figure 2 and 3 display results from the experiments performed to illustrate the basic functionality of uTCP. These experiments simply try to demonstrate uTCP’s working by using possibly exaggerated network settings. The experiments do not represent realistic settings. Figure 2(a) shows the cumulative bytes received over time during the transfer of a 145KB file over TCP and uTCP using a network path with 60 ms round-trip time (RTT) and 3% loss rate. The horizontal lines in TCP curve illustrate TCP behavior during packet loss. During those periods, TCP is waiting for dropped packets and is not delivering any data to the application. Unlike TCP, uTCP steadily delivers data at a constant rate.

Figure 2(b) shows the results of an experiment where a sender sends 30MB of data divided into 1448 bytes messages. This experiment tries to emulate video streaming situation with 2% packet loss rate across a network with 100ms RTT. The graphs show the end-to-end application latency for the 1448 byte messages. The results indicate that with uTCP fewer than 3% of the transmitted messages are significantly delayed. On the other hand, with TCP more than 10% of the messages are delayed by at least one full RTT.
Figure 3 shows the progress of a large file (32 Megabytes) download over uTCP and TCP under two different network conditions: an emulated network with no middleboxes (30ms RTT, 1% loss), and a real 3Mbps DSL connection with a residential NAT in the path (60ms RTT). This figure illustrates that TCP and uTCP behave in an identical manner in a network path that contains a middlebox.

Figure 2: (a) Graph illustrating the working of uTCP. It shows unordered bytes received at the application with uTCP, as compared to TCP's ordered delivery. (b) Graph illustrating uTCP's key benefit to the application: reduction in application-observed latency (Figure from [8])

Figure 3: graph illustrating that uTCP works and performs as well as TCP does under different network scenarios, including NATs. (Figure from [8])
5.2 Bandwidth and CPU Costs

We now present results that show the bandwidth and CPU costs of uTCP and uCOBS. Figure 4 illustrates the results of a 30MB bulk transfer over a path with 60ms RTT. The results show that over a long transfer average throughput is almost identical in all three cases of TCP, COBS and uCOBS. This means that there is no significant bandwidth overhead associated with COBS encoding.

Figure 5 indicates that the COBS encoding and message delivery mechanism has some CPU cost overhead associated with them. The figure shows most of the CPU overhead occurs in the user-space library and not in the kernel. This is expected because we sought to minimize extra processing in the kernel and try doing most of the processing in the user-space. The CPU costs in the user-space result from the queue management functions, message extraction function and COBS decoding function. We believe that further optimizations to user-space code can decrease the CPU costs for uCOBS.

Figure 4: Chart that compares the throughput of TCP, COBS and uCOBS under different loss rates.

Figure 5: CPU costs for TCP, COBS and uCOBS. The lower darker portion of the bars indicate kernel CPU costs and the upper portion of the bars indicate userspace CPU costs.
Section 5: Related Work

SCTP is a good alternative to TCP as a web transport [10]. It allows web applications to use multiple data streams within a single connection to transmit independent objects, thus preventing the head-of-line blocking problem without creating multiple connections. Similarly SST, a new transport protocol that runs on top of UDP also provides multi-streaming facility and is better as a web transport than TCP [6]. However, as mentioned in section 1.3.2, these transports face tremendous deployment hurdles and TCP is here to stay.

Section 6: Conclusions and Future Work

Even though TCP is not suitable for the web applications, these applications are forced to use TCP. These web applications can benefit by using uTCP, our simple modification to how TCP behaves at the end-hosts. Simple modifications can be made to HTTP, the protocol widely used by web applications, to make it work with uTCP. The new SPDY protocol designed by Google as a better alternative to HTTP still uses TCP for a lack of better alternatives [15]. Even SPDY can benefit from using uTCP instead of TCP.

Our modifications to HTTP are incomplete. Particularly, there is currently no mechanism to transfer very large files since they cannot be transferred as one message. This problem can be solved for static files by using already existing HTTP option called "Range" which enables a client to request a huge file in manageable portions. However, the problem remains for transferring large dynamic files, such as dynamically created pdf files. Future work can focus on introducing new options to HTTP to avoid this problem.

We do not specify a mechanism with HTTP in which a server can interleave many web objects together. Interleaving will decrease the client perceived latency especially when the web objects are large. Interleaving can be achieved using uTCP by building a transport protocol such as Stream Control Transport Protocol (SCTP) or Structured Streams (SST) on top of uTCP [6, 13]. We have a simple prototype design showing how to implement the multi-streaming service of SCTP on top of uTCP. Future work can focus on fine adapting the prototype to work with HTTP.
Section 7: References