SCTP: An innovative transport layer protocol for the web

Preethi Natarajan[1], Janardhan R. Iyengar[1], Paul. D. Amer[1] and Randall Stewart[2]

[1]Protocol Engineering Lab, CIS Dept
University of Delaware
{nataraja, iyengar, amer}@cis.udel.edu

[2]Internet Technologies Division
Cisco Systems
rrs@cisco.com

ABSTRACT

We propose using the Stream Control Transmission Protocol (SCTP), a recent IETF transport layer protocol, for reliable web transport. Although TCP has traditionally been used, we argue that SCTP better matches the needs of HTTP-based network applications. This position paper discusses SCTP features that address: (i) head-of-line blocking within a single TCP connection, (ii) vulnerability to network failures, and (iii) vulnerability to denial-of-service SYN attacks. We discuss our experience in modifying the Apache server and the Firefox browser to benefit from SCTP, and demonstrate our HTTP over SCTP design via simple experiments. We also discuss the benefits of using SCTP in other web domains through two example scenarios ─ multiplexing user requests, and multiplexing resource access. Finally, we highlight several SCTP features that will be valuable to the design and implementation of current HTTP-based client-server applications.

Categories and Subject Descriptors

C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks – *Internet*; C.2.6 [**Computer-Communication Networks**]: Internetworking – *Standards*; C.4 [**Performance of Systems**]: Design Studies; Fault Tolerance; Reliability, availability and serviceability.

General Terms

Performance, Design, Security.

Keywords

SCTP, Stream Control Transmission Protocol, fault-tolerance, head-of-line blocking, transport layer service, web applications, web transport.

1. INTRODUCTION

HTTP requires a reliable transport protocol for end-to-end communication. While historically TCP has been used for this purpose, RFC2616 does not require TCP; but until now, no reasonable alternative existed. The Stream Control Transmission Protocol (SCTP), specified in RFC2960, is a recently standardized reliable transport protocol which provides a set of innovative transport layer services unavailable from TCP (or UDP). In this paper, we argue that these services can enhance web transfers, making SCTP a better choice for web transport.

SCTP was originally designed within the IETF SIGTRAN working group to address the shortcomings of TCP for telephony signaling over IP networks [2]. SCTP has since evolved into a general purpose IETF transport protocol, and is well beyond a laboratory research project. More than 25 SCTP implementations currently exist, including kernel implementations for FreeBSD, NetBSD, OpenBSD, Mac OS X, Linux, Solaris, AIX, and HP-UX; and user-space implementations for Windows, on proprietary platforms for Cisco, Nokia, Siemens, and other vendors. Eight interoperability workshops over the past five years have fine-tuned these implementations [14].

Of SCTP's new services and features, SCTP *multistreaming* provides an application with logically separate data streams to transfer multiple independent objects, SCTP *multihoming* provides transparent fault-tolerance to applications on multihomed end hosts, and SCTP's four-way handshake during association (SCTP's term for a connection) establishment avoids denial-of-service SYN attacks. In this paper, we discuss these features and their applicability to web transfers.

The paper is organized as follows. Section 2 details how SCTP solves three specific limitations that occur when HTTP-based client-server applications use TCP: head-of-line blocking, disruption due to network failures, and SYN attacks. Section 3 overviews our modifications to Apache and Firefox architectures to operate over SCTP. We also analyze how their original architectures limit full utilization of SCTP's new features. In Section 4, we explore web domains other than general browsing, and articulate how these domains can benefit from SCTP. Section 5 elaborates other SCTP features and relevant SCTP work that might be useful for HTTP-based network applications. Section 6 summarizes and concludes the paper.

2. HTTP OVER TCP CONCERNS

In this section, we discuss three major concerns in using TCP for web transport, and how our choice ─ SCTP ─ effectively addresses all of these concerns.

2.1 Head of line blocking

Consider the simple case of a web browser displaying a web page. Using HTTP/1.1 that supports persistent and pipelined connections, the browser opens a new transport connection to the server, and sends an HTTP GET request with the desired URI. The server returns an HTTP response with the page contents. This page may contain URIs of embedded objects. The browser parses the content for these URIs, and sends pipelined HTTP GET requests for each of the URIs. As responses arrive from the server, the browser displays the webpage with its embedded objects.

- Prepared through collaborative participation in the Communication and Networks Consortium sponsored by the US Army Research Lab under Collaborative Tech Alliance Program, Coop Agreement DAAD19-01-2-0011. The US Gov't is authorized to reproduce and distribute reprints for Gov't purposes notwithstanding any copyright notation thereon.
- Supported by the University Research Program, Cisco Systems, Inc.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

In general, objects embedded within a web page are *independent* of each other. That is, requesting and displaying each object in the page does not depend on the reception of other embedded objects. This "degree of freedom" is best exploited by *concurrently* downloading and rendering the independent embedded objects.

At the transport layer, TCP offers a single sequential bytestream to an application; all application data are serialized and sent sequentially over the single bytestream. In addition, TCP provides in-order delivery within this bytestream – if a transport protocol data unit (TPDU) is lost in the network, successive TPDUs arriving at the TCP receiver will not be delivered to the application until the lost TPDU is retransmitted and received. Hence, when TCP is used for web transport, a lost TPDU carrying a part of a web object may block delivery of other successfully received independent web objects. This problem, known as head-of-line (HOL) blocking, is due to the fact that TCP cannot logically separate independent application level objects in its transport and delivery mechanisms.

HOL blocking also results in unnecessary filling of the receiver's transport layer buffer space. Reliable transport protocols such as TCP use a receiver buffer to store TPDUs that arrive out-of-order. Once missing TPDUs are successfully retransmitted, data in the receiver buffer is ordered and delivered to the application. This buffer fill up is unnecessary in cases when 'later received' TPDUs belong to a different application object than the earlier lost TPDU(s). The required amount of buffer space increases with the loss probability in the transmission path, and the number of independent objects to be transferred.

Note that HOL blocking is particularly exacerbated in domains with low bandwidth and/or high loss rates. With the proliferation of mobile phones, and the increasing use of web browsers and other web applications on mobile phones, increased HOL blocking will cause significant user-perceived delays.

To alleviate HOL blocking, web browsers usually open multiple TCP connections to the same web server [5]. All HTTP GET requests to the server are distributed among these connections, avoiding HOL blocking between the corresponding responses. However, multiple independent objects transferred within one of the several parallel connections still suffer from HOL blocking.

Using multiple TCP connections for transferring a single application's data introduces many negative consequences for both the application and the network. Previous work such as Congestion Manager [6] and Transaction TCP [17] analyze these consequences in depth, which we summarize:

- *Aggressive behavior during congestion:* TCP's algorithms maintain fairness among TCP (and TCP-like) connections. A TCP sender reduces its congestion window by half when network congestion is detected [13]. This reduction is a well understood and recommended procedure for maintaining stability and fairness in the network [18,19]. An application using multiple TCP connections gets an unfair share of the available bandwidth in the path, since all of the application's TCP connections may not suffer loss when there is congestion in the transmission path. If *m* of the *n* open TCP connections suffer loss, the multiplicative decrease factor for the connection aggregate at the sender is *(1 - m/2n)* [8]. This decrease factor is often greater than one-half, and therefore an application using parallel connections is considered an aggressive sender. This aggressive behavior leads to consumption of an unfair share of the bottleneck bandwidth as compared to applications using fewer connections.
- *Absence of integrated loss detection and recovery:* Web objects are typically small, resulting in just a few TPDUs per HTTP response. In these cases, a TPDU loss is often recoverable only through an expensive timeout at the web server due to an insufficient number of duplicate acks to trigger a fast retransmit [8]. Though this problem is lessened in HTTP/1.1 due to persistent connections and pipelined requests, it still exists while using multiple TCP connections since separate connections cannot share ack information for loss recovery.
- *Increased load on web server:* The web server has to allocate and update a Transmission Control Block (TCB) for every TCP connection. Use of parallel TCP connections between client and server increases TCB processing load on the server. Under high loads, some web servers may choose to drop incoming TCP connection requests due to lack of available memory resources.
- *Increased connection establishment latency:* Each TCP connection goes through a three-way handshake for connection establishment before data transfer is possible. This handshake wastes one round trip for every connection opened to the same web server. Any loss during connection setup can be expensive since a timeout is the only means of loss detection and recovery during this phase. Increasing the number of connections increases the chances of losses during connection establishment, thereby increasing the overall average transfer time.

Congestion Manager (CM) [6] attempts to solve the first two problems. CM is a shim layer between the transport and network layers which aggregates congestion control at the end host, thereby enforcing a fair sending rate when an application uses multiple TCP connections to the same end host. "TCP Session" [7] proposes integrated loss recovery across multiple TCP connections to the same web client (these multiple TCP connections are together referred to as a TCP session). All TCP connections within a session are assumed to share the transmission path to the web client. A Session Control Block (SCB) is maintained at the sender to store information about the shared path such as its congestion window and RTT estimate. Both, CM and TCP Session, still require a web browser to open multiple TCP connections to avoid HOL blocking, thereby increasing the web server's load.

Apart from solving the network related problems due to parallel TCP connections, there has also been significant interest in designing new transport and session protocols that better suit the needs of HTTP-based client-server applications than TCP. Several experts agree (for instance, see [28]) that the best transport scheme for HTTP would be one that supports datagrams, provides TCP compatible congestion control on the entire datagram flow, and facilitates concurrency in GET requests. WebMUX [29] was one such session management protocol that was a product of the (now historic) HTTP-NG working group [30]. WebMUX proposed use of a reliable transport protocol to provide web transfers with "streams" for transmitting independent objects. While the WebMUX effort did not mature, SCTP is a current IETF standards-track protocol with several implementations and a growing deployment base, and offers many of the core features that were desired of WebMUX.

We propose using SCTP's *multistreaming* feature — a previously unavailable transport layer service specifically designed to avoid HOL blocking when transmitting logically independent application objects. An SCTP *stream* is a unidirectional data flow within an SCTP association. Independent application objects can be transmitted in different streams to maintain their logical separation during transfer and delivery. Note that an SCTP association is subject to congestion control similar to TCP. Hence, all SCTP streams within an association are subject to shared congestion control, and thus multistreaming does not violate TCP's fairness principles.

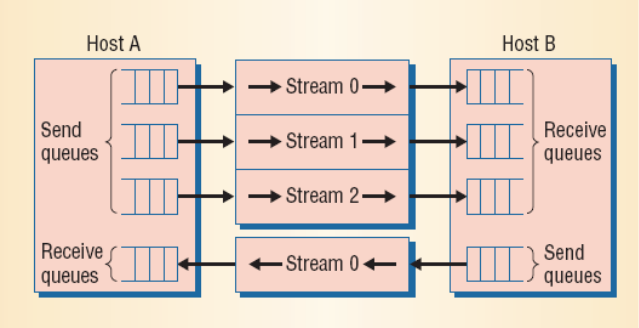

Figure 1. Multistreamed association between two hosts

Figure 1 illustrates a multistreamed association between hosts *A* and *B*. In this example, host *A* uses three *output* streams to host *B* (numbered 0 to 2), and has only one *input* stream from host *B* (numbered 0). The number of input and output streams in an SCTP association is negotiated during association setup.

SCTP uses *stream sequence numbers* (SSNs) to preserve data order within each stream. However, maintaining order of delivery between TPDUs transmitted on different streams is not a constraint. That is, data arriving in-order within an SCTP stream is delivered to the application without regard to data arriving on other streams.

To transfer independent web objects without HOL blocking, each object can be sent in a separate stream, all within a single association. SCTP uses a single global *Transmission Sequence Number* (TSN), which provides integrated loss detection and recovery across streams; loss in one stream can be detected via acks for data on other streams. Also congestion control is shared; a web browser using this solution will be no more aggressive than a web browser using a single TCP connection. Connection establishment latency does not increase with multistreaming. While every association setup requires a four-way handshake, data transfer can begin in the third leg (See Section 2.3).

In KAME SCTP implementation [12][14], the SCTP TCB is approximately twice the size of a TCP TCB. The memory overhead per inbound or outbound stream is 16 bytes, causing the TCB memory requirements for two parallel TCP connections to be roughly equal to the requirements for a single SCTP association with two pairs (inbound and outbound) of streams. However, to achieve higher concurrency, memory overhead when increasing the number of TCP connections is much greater than when increasing the number of streams within an SCTP association.

The size of a TCP TCB is quite high (~700 bytes) when compared to memory overhead for a pair of SCTP streams (32 bytes). Using these values, the memory requirements for the TCP and SCTP cases can be approximated as:

For n parallel TCP connections

$= [n * (\text{TCP TCB size})]$ bytes

$= [n * 700]$ bytes

For 1 SCTP association with n pairs of streams

$= [(\text{SCTP TCB size}) + (n * 32)]$ bytes

$= [(2 * \text{TCP TCB size}) + (n * 32)]$ bytes

$= [1400 + (n * 32)]$ bytes

From the above calculations, it is evident that the memory required for the TCP case increases rapidly with n ($n > 2$) when compared to the SCTP case. Note that with SCTP multistreaming, apart from the lower memory overhead, a web server also incurs the lower processing load of only one TCB per web client.

We discuss a more detailed mapping of HTTP over SCTP, and our implementation of this mapping in Section 3.

2.2 Network Failures

Critical web servers rely on redundancy at multiple levels to provide uninterrupted service during resource failures. A host is multihomed if it can be addressed by multiple IP addresses [4]. Multihoming a web server offers redundancy at the network layer, provided that the web server remains accessible even when one of its IP addresses becomes unreachable, say due to an interface or link failure, severe congestion, or slow route convergence around path outages.

Multihoming end hosts is becoming increasingly economical. For instance, today's relatively inexpensive access to the Internet motivates home users to have simultaneous wired and wireless connectivity through multiple ISPs, thereby increasing the end host's fault tolerance at an economically feasible cost.

TCP is ignorant of multihoming. Even if end hosts have multiple interfaces, an application using TCP cannot leverage this network layer redundancy, since TCP allows the application to bind to only one network address at each end of a connection. For example, in Figure 2, assume that host *A* runs a web server and host *B* runs a web client. Using TCP, the web client can use one interface (*B1*), to connect to one interface (*A1*) at the web server. If *A1* fails, the web server becomes unreachable to all the clients connected through *A1*, including *B*, and the corresponding TCP connections are aborted. Unfortunately, the redundant active network interface, *A2*, could not be used by the clients connected through *A1*.

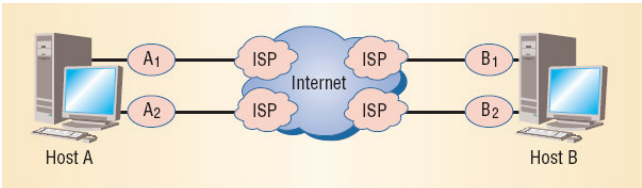

Figure 2. Multihomed end hosts

To provide applications on multihomed end hosts with resilience to such failures, SCTP supports *multihoming* — a transport layer

feature providing transparent network failure detection and recovery. SCTP allows binding a transport layer association to multiple IP addresses at each end host. An end point chooses a single primary destination address for sending new data. SCTP monitors the reachability of each destination address through two mechanisms: acks of data and periodic probes known as *heartbeats*. Failure in reaching the primary destination results in *failover*, where an SCTP endpoint dynamically chooses an alternate destination to transmit the data, until the primary destination becomes reachable again.

In Figure 2, a single SCTP association is possible between addresses *A1*, *A2* at the server and *B1*, *B2* at the client. Assuming *A1* is the primary destination for the client, if *A1* becomes unreachable, multihoming keeps the SCTP association alive through failover to alternate destination A2, and allows the end host applications to continue communicating seamlessly.

Ongoing research on Concurrent Multipath Transfer (CMT) [15], proposes to use multihoming for parallel load sharing. During scenarios where multiple active interfaces between source and destination connect through independent paths, CMT simultaneously uses these multiple paths to transfer new data, increasing throughput for a networked application. Thus, a multihomed web client and server running on SCTP can leverage CMT's throughput improvements for web transfers.

2.3 SYN Attacks

A SYN attack is a common denial of service (DoS) technique that has often disabled the services offered by a web server. During the three-way TCP connection establishment handshake, when a TCP server receives a SYN, the TCP connection transitions to the TCP half-open state. In this state, the server allocates memory resources, stores state for the SYN received, and replies with a SYN/ACK to the sender. The TCP connection remains half open until it receives an ACK for the SYN/ACK resulting in connection establishment, or until the SYN/ACK expires with no ACK. However, the latter scenario results in unnecessary allocation of server's resources for the TCP half open connection.

When a malicious user orchestrates a coordinated SYN attack, 1000's of malicious hosts flood a predetermined TCP server with IP-spoofed SYN requests, causing the server to allocate resources for many half open TCP connections. The server's resources are thus held by these fabricated SYN requests, denying resources to legitimate clients. Such spoofed SYN attacks are a significant security concern, and an inherent vulnerability with TCP's three-way handshake. Web administrators try to reduce the impact of such attacks by limiting the maximum number of half open TCP connections at the server, or through firewall filters that monitor the rate of incoming SYN requests.

To protect an end host from such SYN attacks, SCTP uses a *four-way* handshake with a *cookie* mechanism during association establishment. The four-way handshake does not increase the association establishment latency, since data transfer can begin in the third leg. As shown in Figure 3, when host *A* initiates an association with host *B*, the following process ensues:

1. *A* sends an INIT to *B*.

2. On receipt of the INIT, *B* does not allocate resources to the requested association. Instead, *B* returns an INIT-ACK to *A* with a cookie that contains: (i) necessary details required to identify and process the association (ii) life span of the cookie, and (iii) signature to verify the cookie's integrity and authenticity.

3. When *A* receives the INIT-ACK, *A* replies with a COOKIE-ECHO, which echoes the cookie that *B* previously sent. This COOKIE-ECHO may carry *A*'s application data to *B*.

4. On receiving the COOKIE-ECHO, *B* checks the cookie's validity, using the state information in the cookie. If the cookie verifies, *B* allocates resources and establishes the association.

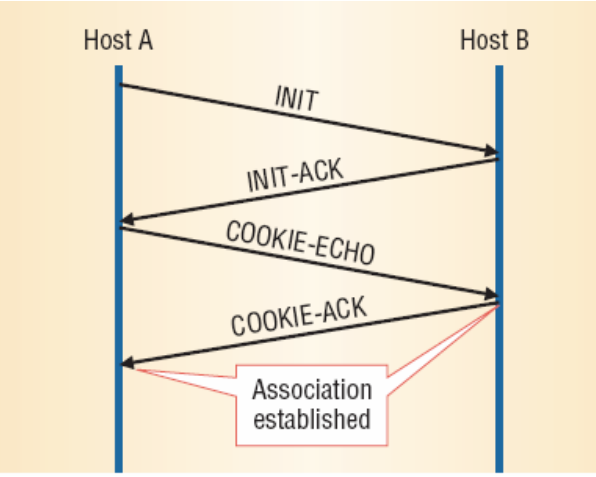

Figure 3. SCTP association establishment

With SCTP's four-way handshake, a web client that initiates an association must maintain state before the web server does, avoiding spoofed connection request attacks.

3. APACHE AND FIREFOX OVER SCTP

To investigate the viability of HTTP over SCTP, we modified Apache and Firefox to run over SCTP in FreeBSD 5.4 [12]. These modified implementations are publicly available [20]. In this section, we list our design guidelines, and discuss the rationale behind our final design. We then present the relevant architectural details of Apache and Firefox, and describe our changes to their implementation. Finally, we discuss their architectural limitations which do not allow the application to fully benefit from SCTP multistreaming. These limitations are possibly shared by other web servers and browsers as well.

3.1 Design Guidelines

Two guidelines that governed our HTTP over SCTP design were:

- Make no changes to the existing HTTP specification, to reduce deployment concerns
- Minimize SCTP-related state information at the server so that SCTP multistreaming does not become a bottleneck for performance.

An important design question to address was: which end (the client or server) should decide on the SCTP stream to be used for an HTTP response? Making the web server manage some form of SCTP stream scheduling is not desirable, as it involves maintaining additional state information at the server. Further, the client is better positioned to make scheduling decisions that rely on user perception and the operating environment. We therefore concluded that the client should decide object scheduling on streams.

We considered two designs by which the client conveys the selected SCTP stream to the web server: (1) the client specifies the stream number in the HTTP GET request and the server sends the corresponding response on this stream, or (2) the server transmits the HTTP response on the same stream number on which the corresponding HTTP request was received. Design (1) can use just one incoming stream and several outgoing streams at the server, but requires modifications to the HTTP GET request specification. Design (2) requires the server to maintain as many incoming streams as there are outgoing streams, increasing the memory overhead at the server. The KAME SCTP TCB uses 16 bytes for every inbound or outbound stream. We considered this memory overhead per stream to be insignificant when compared to changes to HTTP specification, and chose option (2).

3.2 Apache

We chose the Apache (version 2.0.55) open source web server for our task. In this section, we give an overview of Apache's architecture, and their modifications to use SCTP streams.

3.2.1 Architecture

The Apache HTTP server has a modular architecture. The main functions related to server initialization, listen/accept connection setup, HTTP request parsing, memory management are handled by the *core* module. The remaining accessory functions such as request redirection, authentication, dynamic content handling are performed by separate modules. The core module relies on the Apache Portable Runtime (APR), a platform independent API, for network, memory and other system dependent functions.

Apache has a set of multi-processing architectures that can be enabled during compilation. We considered the following architectures: (1) *prefork* ─ non-threaded pre-forking server and (2) *worker* ─ hybrid multi-threaded multi-processing server. With *prefork*, a configurable number of processes are forked during server initialization, and are setup to listen for connections from clients. With *worker*, a configurable number of server threads and a listener thread are created per process. The listener thread listens for incoming connections from clients, and passes the connection to a server thread for request processing.

In both architectures, a *connection* structure is maintained throughout a transport connection's lifetime. Apache uses *filters* ─ functions through which different modules process an incoming HTTP request (input filters) or outgoing HTTP response (output filters). The core module's input filter calls the APR *read* API for reading HTTP requests. Once the HTTP request syntax is verified, a *request* structure is created to maintain state related to the HTTP request. After processing the request, the core module's output filter calls the APR *send* API for sending the consequent HTTP response.

3.2.2 Changes

To adapt Apache to use SCTP streams, the APR *read* and *send* API implementations were modified to collect the SCTP input stream number on which a request is read, and to send the response on the corresponding output stream. During a request's lifetime, a temporary storage place stores the SCTP stream number for the request. The initial design was to use the *socket* or *connection* structures for the purpose. But, pipelined HTTP requests from potentially different SCTP streams can be read from the same socket or connection, overwriting previous information. Hence these structures were avoided. In our implementation, stream information related to an HTTP request is stored in the *request* structure, and is exchanged between the APR and the core module through Apache's storage buffers (*bucket brigades*).

Apache uses a configuration file that allows users to specify various parameters. We made changes to the *Listen* directive syntax in the configuration file so that a web administrator can specify the transport protocol – TCP or SCTP, to be used by the web server.

3.3 Firefox

We chose the Firefox (version 1.6a1) browser since it is a widely used open-source browser. In this section, we briefly discuss Firefox's architecture, and its adaptation to work over SCTP streams.

3.3.1 Architecture

Firefox belongs to the Mozilla suite of applications which have a layered architecture. A set of applications, such as Firefox and Thunderbird (mail/news reader), belong to the top layer. These applications rely on the services layer for access to network services. The services layer uses platform independent network APIs offered by the Netscape Portable Runtime (NSPR) library in the runtime layer. NSPR maintains a *methods* structure with function pointers to various I/O and other management functions for TCP and UDP sockets.

Firefox has a multi-threaded architecture. To render a web page inside a Firefox *tab,* first the HTTP protocol handlers parse the URL, and use the socket services to open a TCP connection to the web server. Once the TCP connection is setup, an HTTP GET request for the web page is sent. After the web page is retrieved and parsed, further HTTP requests for embedded objects are pipelined over the same TCP connection if the connection persists; else over a new TCP connection.

In the version we used, Firefox never opened more than one TCP connection for a *simple* transaction to the same web server. However, when we requested multiple news-feeds from the same web server in different tabs (*"Open all in tabs"* feature, where multiple pages are displayed concurrently), Firefox opened multiple TCP connections to the same web server, one for each tab.

3.3.2 Changes

Adapting Firefox to work on SCTP streams involved modifications in its services layer to open an SCTP socket instead of a TCP socket, and creating a new *methods* structure in NSPR for SCTP related I/O and management functions.

During SCTP association setup with the server, Firefox requests a specific number of SCTP input and output streams. (In SCTP, this request can be negotiated down by the server in the INIT-ACK.) For our purposes, the number of input streams is set to equal the number of output streams, thus assuring that the Firefox browser receives a response on the same stream number as the one on which it sends a request.

Our Firefox changes provide flexibility to do HTTP request scheduling over SCTP streams. The current implementation picks SCTP streams in a round-robin fashion. Other scheduling approaches can be considered in the future. For example, in a lossy network environment, such as wide area wireless connectivity through GPRS, a better scheduling policy might be 'smallest pending object first' where the next GET request goes on the SCTP stream that has the smallest sum of object sizes pending transfer. Such a policy reduces the probability of HOL

blocking among the response for the most recent GET request and the responses for previous requests transmitted on the same SCTP stream.

With Firefox's current design, the choice of the transport protocol (TCP or SCTP) must be decided at compile time. In the future, it will be beneficial to have this choice as a configurable parameter.

3.4 SCTP Multistreaming Avoids HOL

We present two simple experiments to visualize the differences between the current HTTP over TCP design, and our HTTP over SCTP multistreaming design. Our goal is to demonstrate how HTTP over SCTP multistreaming avoids HOL blocking.

The experiment topology, shown in Figure 4, uses three nodes: a custom web client (FreeBSD 5.4) and an Apache server (FreeBSD 5.4) connected by Dummynet (FreeBSD 4.10) [24]. Dummynet's traffic shaper configures a 56Kbps duplex link, with a queue size of 50KB and zero added propagation delay between client and server. This link has no loss in the direction from client to server, and 10% loss from server to client.

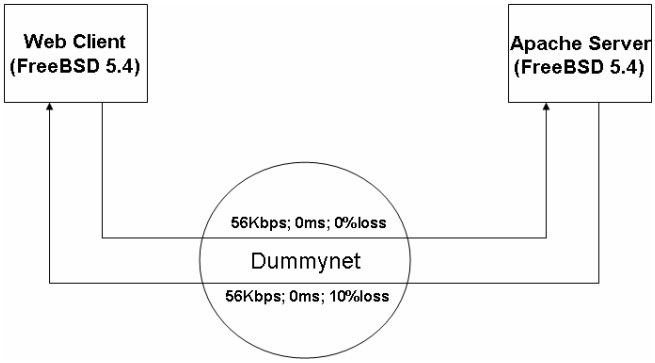

Figure 4. Experiment Topology

In both experiments, the client requests a web page containing 5 embedded 5.5KB objects (for example, a photo album page containing 5 embedded JPEG images) from the Apache server. In the first experiment, the web client and Apache communicate over a single TCP connection, and, in the second they communicate over a single SCTP association with one stream for each embedded object.

Using timestamp information collected from tcpdump [25] traces at the client, Figures 5 and 6 plot PDU receipt times at the transport and application layers in the TCP and SCTP runs, respectively. A point labeled '*n*' denotes the arrival of one of object *n*'s TPDUs at the receiving transport layer. A corresponding 'X' denotes the earliest *calculated* time when the data in that TPDU is delivered by the transport layer to the application.

In both scenarios, TPDU 6 (2nd TPDU of object 2) is lost, and its retransmission arrives just after time=4 seconds. This loss causes the remaining TPDUs to arrive 'out-of-order' at the client's transport layer. In HTTP over TCP (Figure 5), HOL blocking by object 2, causes TCP to delay delivery of data in objects 3, 4 and 5 until the successful retransmission of TPDU 6. Note that even after this retransmission, TCP is still blocked from delivering object 5 to the application due to loss of TPDU 19 (5th TPDU of 4th object). In HTTP over SCTP (Figure 6), the TPDUs for each object arrive on a different SCTP stream. Hence, the loss[1] of TPDU 6 – object 2's TPDU, does not block application delivery of objects 3, 4 or 5. Note that in Figure 6, the initial four PDUs of object 4 are delivered without HOL blocking. The final TPDU of object 4 is lost and is delivered only after the retransmission arrives.

We believe that SCTP multistreaming and the absence of HOL blocking opens up opportunities for a new range of browser and server features, which we discuss in detail in the following sections.

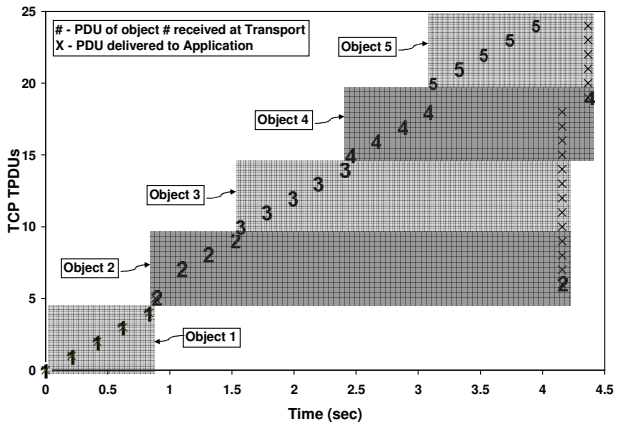

Figure 5. HOL blocking in HTTP over TCP

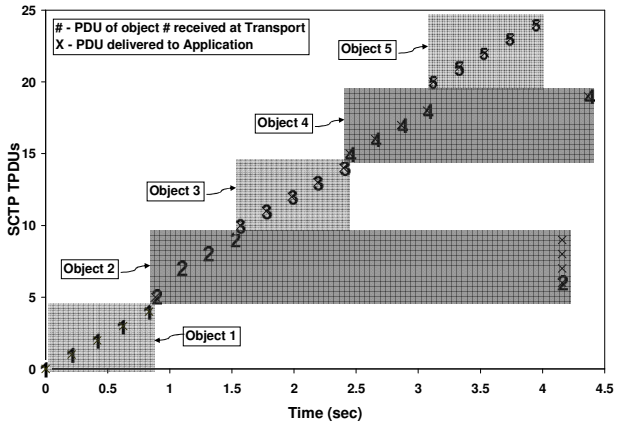

Figure 6. No HOL blocking in HTTP over SCTP

3.5 Browser/Server Architectural Discussion

Even if SCTP can deliver requests and responses of independent web objects without HOL blocking, the current Apache and Firefox architectures are unable to take full advantage of SCTP's multistreaming benefits. We explain a browser side architectural limitation, and propose a solution. We also explain a server side

[1] In our SCTP experiment, each application write generated an SCTP TPDU, causing a one to one correspondence between a TPDU's Transmission Sequence Number (TSN) and the TPDU number.

architectural change that can enhance the server's performance, especially in lossy environments.

3.5.1 Browser Limitation

If SCTP has received partial data for *n* independent web objects on different streams, SCTP will deliver these *n* partial responses to the web browser as long as TPDUs within each stream arrived in sequence. A web browser now has the opportunity to read and render these *n* responses *concurrently*. This browser capability, known as *parallel rendering*, can be optionally used to improve user perception since multiple web objects start appearing in parallel on the corresponding web page.

Parallel rendering is difficult to realize with the current Firefox architecture. Firefox dedicates a single thread to a transport layer connection. This design reflects Firefox's assumption regarding TCP as the underlying transport. With TCP, objects can be received only sequentially within a single connection; hence a single thread to read the HTTP responses is sufficient. In the modified implementation, a single thread gets dedicated to an SCTP association. Consequently, the thread sends the pipelined HTTP GET requests, and reads the responses in sequence. Therefore, multiple streams within an SCTP association are still handled *sequentially* by the thread, allowing the thread to render at most one response at a time.

One possible solution to realize parallel rendering in Firefox (or any multi-threaded web browser) is to use *multiple* threads to request and render web page objects via *one* SCTP association. Multiple threads, one for each object, send HTTP GET requests over different SCTP streams of the association. The number of SCTP streams to employ for a web page can be either user configurable, or dynamically decided by the browser. The same thread that sends the request for an object can be responsible for rendering the response. However, it is necessary that a single 'reader' thread reads all the HTTP responses for a web page, since TPDUs from a web server containing the different responses can arrive interleaved at the browser's transport layer (discussed in Section 3.6).

Multiple threads enable parallel rendering but require considerable changes to Firefox's architecture. We suspect most common web browsers to suffer from a similar architectural limitation.

3.5.2 Server Enhancement

The original multi-threaded Apache dedicates one server thread to each TCP connection. Our adaptation over SCTP multistreaming dedicates a server thread to an SCTP association. In this design, the server thread reads HTTP requests *sequentially* from the association, even if requests arrive on different streams in the association.

Apache might achieve better concurrency in serving user requests if its design enabled multiple threads to read from different SCTP streams in an association, each capable of delivering independent requests without HOL blocking. We hypothesize that in lossy and/or low bandwidth environments, this design can provide higher request service rates when compared to Apache over TCP, or our current Apache over SCTP.

3.6 Object Interleaving

In this section, we use "imaginary" scenarios to illustrate *object interleaving*. Object interleaving ensues when a browser and a server, capable of transmitting HTTP requests and responses *concurrently*, communicate over different streams of an SCTP association. For example, object interleaving will be observed when a multi-threaded browser and server, modified as described in Section 3.5, communicate over SCTP streams. Since such browser and server implementations are in progress, we use imaginary data to illustrate the concept.

We use two scenarios in our demonstration. Each scenario shows one of the two extreme cases ─ the presence of an ideal object interleaving, and no interleaving. In both scenarios, a multi-threaded browser requests 5 objects from a multi-threaded web server. Every object is the same size and is distributed over 5 TPDUs, resulting in a total of 25 TPDUs for each transfer. The transfers do not experience any loss or propagation delay. The transmission time for each TPDU is around 180ms, resulting in a total transfer time of ~4.4 seconds.

In the first scenario, the multi-threaded browser and server are adapted as discussed in Section 3.5. The browser uses 5 threads to send GET requests concurrently on 5 SCTP streams. Due to this concurrency, the GET requests get bundled into SCTP TPDUs at the browser's transport layer. For our illustration, we consider an ideal bundling where all 5 requests get bundled into one TPDU. When this TPDU reaches the server's transport layer, multiple server threads concurrently read the 5 requests from SCTP and send responses. The concurrency in sending responses causes TPDUs containing different objects to get interleaved at the server's, and hence the browser's transport layer, causing object interleaving. Note that the degree of object interleaving depends on (1) the browser's request writing pattern, which dictates how requests get bundled into SCTP TPDUs at browser's transport layer, and (2) the sequence in which the server threads write the responses for these requests.

For the second scenario, the multi-threaded browser and server are adapted to use SCTP multistreaming, but do not have the necessary modifications to concurrently send requests or responses. The browser uses a single thread to sequentially send the 5 GET requests over 5 SCTP streams. Each request gets translated to a separate SCTP PDU at the browser's transport layer. These 5 SCTP PDUs, and hence the 5 HTTP requests arrive in succession at the web server, which uses a single thread to read and respond to these requests. These responses arrive sequentially at the browser's transport layer.

Figure 7 illustrates the ideal object interleaving at the browser's transport layer, where the first 5 TPDUs are the first TPDUs of all the 5 responses. The next 5 TPDUs correspond to the second TPDUs of the 5 responses and so on. Figure 8 illustrates the scenario of no object interleaving, and shows how TPDUs corresponding to the 5 responses are delivered one after the other to the browser.

A browser can optionally take advantage of object interleaving to progressively render these 5 objects in parallel, vs. complete rendering of each object in sequence. For example, a browser can render a piece of all 5 objects by time=0.75 seconds (Figure 7) vs. complete rendering of object 1 (Figure 8). By time=2.75 seconds, more than half of all 5 objects can be rendered in parallel with object interleaving vs. complete rendering of objects 1 through 3 in case of no interleaving. The dark and the light rectangles in Figure 7, help visualize the interleaving, and thus the progressive appearance of objects 2 and 4 on a web page.

Apart from progressive parallel rendering in web browsers, HTTP-based network applications can take advantage of object interleaving in other possible ways. For example, if a critical web

client can make better decisions using progressive pieces of all responses vs. complete responses arriving sequentially, the web application's design can gain from object interleaving.

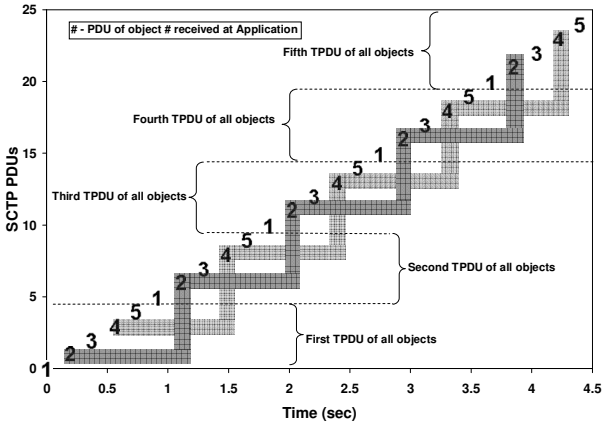

Figure 7. HTTP over SCTP with object interleaving

.

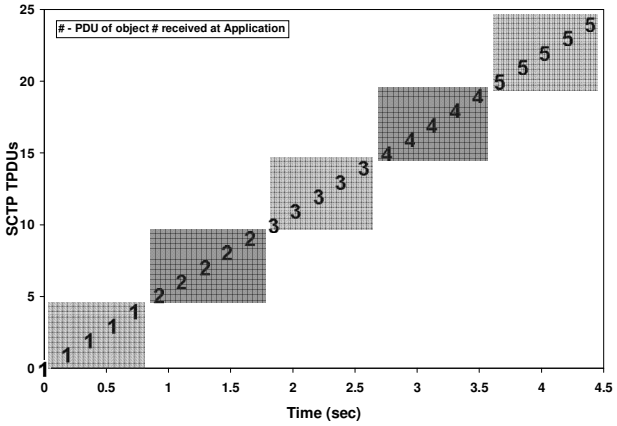

Figure 8. HTTP over SCTP without object interleaving

We point out that object interleaving between a browser and server communicating over TCP is infeasible without explicit application level markers that differentiate TPDUs belonging to different interleaved objects. We feel that such markers try to emulate SCTP multistreaming at the application layer. Also, loss of a single TCP PDU in an interleaved transfer exacerbates the HOL blocking since the loss blocks application delivery of multiple objects.

Browser architectures that facilitate object interleaving can be designed such that the browser is able to control the amount of interleaving for each web transfer. For example, Section 3.5 modifications to a multi-threaded browser will empower it with such flexibility as follows. If the browser uses a single thread to send GET requests *sequentially* on different SCTP streams, the responses will arrive without any interleaving, as shown in Figure 8. On the other hand, if the browser uses multiple threads to send the requests *concurrently* on different SCTP streams, the TPDUs will arrive interleaved as shown in Figure 7. With this flexibility, the web browser can make on-the-fly decisions about how much object interleaving to beget for each web transfer based on prior knowledge about the type of objects being transferred. Such knowledge can be either implicit or explicitly obtained from the web server.

4. OTHER MULTISTREAMING GAINS

We now consider two other web scenarios where SCTP multistreaming might provide a better solution than existing TCP-based solutions.

4.1 Multiplexing User Requests

Several web server farms and providers of Internet service use TCP connection multiplexers to improve efficiency [16]. The main goal of these multiplexers is to decrease the number of TCP connection requests to a server, and thereby reduce server load due to TCP connection setup/teardown and state maintenance. The multiplexer, acting as an intermediary, intercepts TCP connection open requests from different clients, and multiplexes HTTP requests from different clients onto a set of existing TCP connections to the server.

In this scenario, a multiplexer is forced to maintain several open connections to its web server to avoid HOL blocking between independent users' requests and responses. Hence, a tradeoff exists in deciding the number of open connections ─ fewer connections decrease the server load on connection maintenance, whereas more connections reduce HOL blocking between different users' requests.

SCTP multistreaming can be leveraged to reduce both HOL blocking and server load in such an environment. A proxy in front of an SCTP-capable web server can intercept incoming SCTP association open requests from different users. This proxy can maintain just one SCTP association to the web server, and can channel incoming requests from different users on different SCTP streams within this association. Since SCTP multistreaming avoids HOL blocking, this solution is equivalent to having a separate session or connection per user. This setup incurs minimal resource consumption at the server since all data between proxy and server go over a single SCTP association. This design also takes advantage of integrated congestion management and loss recovery within the SCTP association (Section 2.1).

There could be scenarios where a web server runs on SCTP to take advantage of its many features, but a web browser does not have SCTP support. To facilitate seamless service to such browsers, we can extend the multiplexing proxy to act as an application level gateway between HTTP-over-TCP and HTTP-over-SCTP implementations. The proxy can intercept TCP connection open requests, multiplex user requests on different streams of a single SCTP association to the server, and forward server responses to the clients on TCP. This setup ensures the benefits of SCTP multistreaming at the server side, even when the web clients are not SCTP-aware.

4.2 Multiplexing Resource Access

Today's web servers deliver much more to users than just browsing content. For example, business services such as financial planning and tax preparation are offered over the web, and the user accesses these services through a web browser. There are also *web applications* such as online games and web-based mail that are accessible by a browser. In such web applications, a user first establishes a session with the server, and the bulk of the user's data is stored and processed at the server.

Most organizations rely on third-party data centers to host and maintain their web-based software services. For load sharing and better performance, a data center might employ various scheduling policies to logically group and host many web applications on a server. Consider a policy where multiple web applications that will be accessed by the business clients or employees of a single organization are grouped and hosted on the same web server. For example, the data center might host an organization's customer relationship management software and its mail server on the same web server. In such a case, the employees of the organization will access the two resources concurrently from the web server. Instead of opening separate TCP connections for each resource, the user's browser and the web server can multiplex the resource access on different streams of a single SCTP association, reducing load at the server.

5. OTHER USEFUL SCTP FEATURES

Apart from multistreaming, multihoming and protection from SYN attacks, we present other features and related work on SCTP which we believe could be useful to HTTP-based network applications or web applications.

- *Preservation of message boundaries:* SCTP offers a message-oriented data transfer to an application, as opposed to TCP's byte stream data transfer. SCTP considers data from each application *write* as a separate message. This message's boundary is preserved since SCTP guarantees delivery of a message in its entirety to a receiving application. Web applications where the client and server exchange data as messages can benefit from this feature, and avoid using explicit application level message delimiters.
- *Partial Reliability:* RFC3758 describes PR-SCTP, a partial reliability extension to RFC2960. This extension enables partially reliable data transfer between a PR-SCTP sender and receiver. In TCP, and plain SCTP, all transmitted data are guaranteed to be delivered. Alternatively, PR-SCTP gives an application the flexibility to notify how persistent the transport protocol should be in trying to deliver a particular message, by allowing the application to specify a "lifetime" for the message. A PR-SCTP sender tries to transmit the message during this lifetime. Upon lifetime expiration, a PR-SCTP sender discards the message irrespective of whether or not the message was successfully transmitted. This timed reliability in data transfer might be useful to web applications that regularly generate new data obsolescing earlier data, for example, an online gaming application, where a player persistently generates new position coordinates. A game client can use PR-SCTP, and avoid transmitting the player's older coordinates when later ones are available, thereby reducing network traffic and processing at the game server.
- *Unordered data delivery:* SCTP offers unordered data delivery service. An application message, marked for unordered delivery, is handed over to the receiving application as soon as the message's TPDUs arrive at the SCTP receiver. Since TCP preserves strict data ordering, using a single TCP connection to transmit both ordered and unordered data results in unwanted delay in delivering the unordered data to the receiving application. Hence, applications such as online game clients that need to transmit both ordered and unordered data open a TCP connection for the ordered data, and use a separate UDP channel to transmit the unordered data [23]. These applications can benefit from SCTP by using a single SCTP association to transmit both types of data. As opposed to UDP's best effort transmission, which burdens the application to implement its own loss detection and recovery, messages can be transmitted reliably using SCTP's unordered service.
- *SCTP shim layer:* To encourage application developers and end users to widely adopt SCTP and leverage its benefits, a TCP-to-SCTP shim layer has been developed [22]. The shim is a proof of concept and translates application level TCP system calls into corresponding SCTP calls. By using such a shim layer, a legacy TCP-based web application can communicate using SCTP without any modifications to the application's source code.

6. CONCLUSION

Though SCTP has TCP-like congestion and flow control mechanisms targeted for bulk data transfer, we argue that SCTP's feature-set makes it a better web transport than TCP. Performance-wise, SCTP's multistreaming avoids TCP's HOL blocking problem when transferring independent web objects, and facilitates aggregate congestion control and loss recovery. Functionality-wise, SCTP's multihoming provides fault-tolerance and scope for load balancing, and a built-in cookie mechanism in SCTP's association establishment phase provides protection against SYN attacks.

We shared our experiences in adapting Apache and Firefox for SCTP multistreaming, and demonstrated the potential benefits of HTTP over SCTP streams. We also presented current architectural limitations of Apache and Firefox that inhibit them from completely realizing the benefits of multistreaming.

We discussed other systems on the web where SCTP multistreaming may be advantageous, and hypothesized the potential gains of using SCTP in such areas. We also outlined other relevant SCTP features that are useful to HTTP based network applications.

The authors hope that this position paper raises interest within the web community in using SCTP as the transport protocol for web technologies, and welcome further research and collaboration along these lines.

7. ACKNOWLEDGMENTS

The authors thank Armando L. Caro Jr. (BBN Technologies), Ethan Giordano, Mark J. Hufe, and Jonathan Leighton (University of Delaware's Protocol Engineering Lab), and the reviewers of WWW2006 for their valuable comments and suggestions.

8. REFERENCES

[1] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, V. Paxson, "Stream Control Transmission Protocol," RFC 2960, 10/00

[2] R. Stewart, Q. Xie, *Stream Control Transmission Protocol (SCTP): A Reference Guide*, Addison Wesley, 2001, ISBN: 0-201-72186-4

[3] R. Fielding et al., "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, 6/99

[4] R. Braden, "Requirements for Internet hosts – communication layers," RFC1122, 10/89

[5] Z. Wang, P. Cao, "Persistent connection behavior of popular browsers," Research Note, 12/98, www.cs.wisc.edu/~cao/papers/persistent-connection.html

[6] H. Balakrishnan, H.S. Rahul, S. Seshan, "An integrated congestion management architecture for Internet hosts," ACM SIGCOMM, Cambridge, 8/99

[7] V. N. Padmanabhan, "Addressing the challenges of web data transport," PhD Dissertation, Comp Sci Division, U Cal Berkeley, 9/98

[8] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, R. Katz, "TCP behavior of a busy Internet server: Analysis and Improvements," IEEE INFOCOM, San Francisco, 3/98

[9] The Apache Software Foundation, www.apache.org

[10] Netcraft Web Server Survey, news.netcraft.com/archives/web_server_survey.html

[11] Mozilla Suite of Applications, www.mozilla.org

[12] The KAME Project, www.kame.net/

[13] V. Jacobson, "Congestion avoidance and control," ACM SIGCOMM, Stanford, 8/88

[14] Stream Control Transmission Protocol, www.sctp.org/

[15] J. Iyengar, P. Amer, R. Stewart, "Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths," IEEE/ACM Trans on Networking (to appear)

[16] Accelerated Traffic Management, Array Networks, www.arraynetworks.net/products/TMX1100.asp

[17] R. Braden, "Transaction TCP - Concepts," RFC 1379, 9/92

[18] D. M. Chiu, R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," Computer Networks and ISDN Systems, 17(1):1-14, 6/89

[19] M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control," RFC 2581, 4/99

[20] Protocol Engineering Lab, U Delaware, URL: www.pel.cis.udel.edu/

[21] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, P. Conrad, "Stream Control Transmission Protocol (SCTP) Partial Reliability Extension," RFC 3758, 5/04

[22] R. Bickhart, "SCTP shim for legacy TCP applications", MS Thesis, Protocol Engineering Lab, U Delaware, 8/05

[23] Blizzard Entertainment, Technical Support Site, URL: www.blizzard.com/support/

[24] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," ACM CCR, 27(1), 1/97

[25] TCPDUMP Public Repository, www.tcpdump.org/

[26] PCWorld.com – Firefox Downloads Top 100 Million, URL: www.pcworld.com/news/article/0,aid,123140,00.asp

[27] D. Reed, email to end2end-interest mailing list, 10/02. URL: www.postel.org/pipermail/end2end-interest/2002-October/002434.html

[28] J. Gettys, email to end2end-interest mailing list, 10/02. URL: www.postel.org/pipermail/end2end-interest/2002-October/002436.html

[29] J. Gettys, H. Nielsen, "The WebMUX Protocol," URL: www.w3.org/Protocols/MUX/WD-mux-980722.htm

[30] HTTP-NG working group (historic). URL: www.w3.org/Protocols/HTTP-NG/