DHTTP: An Efficient and Cache-Friendly Transfer Protocol for Web Traffic

Michael Rabinovich
AT&T Labs – Research
misha@research.att.com

Hua Wang
New York University
wanghua@cs.nyu.edu

Abstract— **Today's Web interactions are frequently short, with an increasing number of responses carrying only control information and no data. While HTTP uses client-initiated TCP for all Web interactions, TCP is not always well-suited for short interactions. Furthermore, client-initiated TCP handicaps the deployment of *interception caches* in the network because of the possibility of disrupted connections when some client packets bypass the cache on their way to the server.**

We propose a new transfer protocol for Web traffic, called Dual-transport HTTP (DHTTP), which splits the traffic between UDP and TCP channels. When choosing the TCP channel, it is the server who opens the connection back to the client. Among important aspects of DHTTP are adapting to bottleneck shifts between a server and the network and coping with the unreliable nature of UDP. The comparative performance study of DHTTP and HTTP using trace-driven simulation as well as testing real HTTP and DHTTP servers showed a significant performance advantage of DHTTP when the bottleneck is at the server and comparable performance when the bottleneck is in the network. By using server-initiated TCP, DHTTP also eliminates the possibility of disrupted TCP connections in the presence of interception caches thereby allowing unrestricted caching within backbones.

I. Introduction

HTTP was conceived as essentially a protocol for transferring files. A logical consequence was to design it on top of a connection-oriented transport protocol such as TCP. At the same time, the current Web workload exhibits a large number of short page transfers and interactions for control purposes rather than data transfers. For example, in a trace of a large number of modem users [1], 26% of all interactions were cache validations that resulted in a "not-modified" response. Arlitt et al. observed that even for high-speed cable modem users (who intuitively would be more likely to access larger objects) and even considering only responses that did carry data ("successful" responses with 200 response code), the median response size was just 3,450 bytes [2]. Median response sizes of 1.5-3KB were also reported in numerous earlier studies, e.g., [3], [4], [5].

Such behavior is not always served well by TCP. In HTTP 0.9, each Web download paid a TCP connection establishment overhead. Later versions of HTTP address these overheads by introducing *persistent connections* and *pipelining* [6], [7]. Persistent connections allow a client to fetch multiple pages from the same server over the same TCP connection, amortizing the TCP set-up overhead. Pipelining lets the client send multiple requests over the same connection without waiting for responses. The server will send a stream of responses back.

These features have been shown to reduce client latency and network traffic [8]. However, they do not eliminate all overheads of TCP, and in fact may introduce new performance penalties, especially when the bottleneck is at the server [9]. Persistent connections increase the number of open connections at the server, which can have a significant negative effect on server throughput. Pipelining has a limitation that servers must send responses in their entirety and in the same order as the order of the requests in the pipeline. This constraint causes *head of line* delays when a slow response holds up all other responses in the pipeline. To avoid head of line delays, browsers often open multiple simultaneous connections to the same server, further increasing the number of open connections and degrading the throughput of a busy server (see [9] and Section V-B).

To limit the number of open connections, servers close connections that remain idle for a *persistent connection timeout* period. Busy sites often use short connection timeouts, thereby limiting the benefits of persistent connections (see Section V-A). Moreover, persistent connections that servers do maintain are often underutilized, which wastes server resources and hurts the connection's ability to transmit at proper rate (since well-behaving TCP implementations shut down the transmission windows of idle connections [10]).

The Dual-Transport HTTP protocol (DHTTP) described in this paper splits Web traffic between UDP and TCP. A DHTTP client typically sends all requests by UDP. The server sends its response over UDP or TCP, depending on the size of the response and the network conditions. By using UDP for short responses, DHTTP reduces *both* the number of TCP connection set-ups *and* the number of open connections at the server. Also, the utilization of the remaining TCP connections increases because they are reserved for larger objects. Finally, DHTTP does not have the ordering constraints of pipelining.

Furthermore, when choosing TCP, a DHTTP server establishes the connection back to the client, reversing in a sense the client/server roles in the interaction. While having some implications with firewalls (see Section IV-B), this role reversal brings major benefits. First, it avoids an increase (compared to the current HTTP) in the number of message round-trips before the client starts receiving the data over the TCP channel (see Section III). (Of course, when the server uses UDP, the number of message round-trips decreases.) Second, it removes a bottleneck process at the server that accepts all TCP connections. Third, as explained below, it allows unconstrained deployment of *interception caches* in the network.

Interception caches intercept client requests on their path to origin servers and respond to clients on servers' behalf. Interception caching is attractive to ISPs because it occurs transparently to clients and thus relieves ISPs of the administrative burden of configuring client browsers; in fact, ISPs often do not even know or have control over the end-users, as their immediate clients may actually be other ISPs or corporate networks. On the other hand, interception caches use servers' IP addresses when responding to clients thereby impersonating the origin servers and violating the end-to-end semantics of the Internet. In particular, this places significant limits on where interception caches can be deployed and which clients they can serve (see Section IV-A).

In contrast to existing HTTP, DHTTP interception caches use their true IP addresses in their communication with clients. Thus, DHTTP retains the end-to-end semantics of the Internet even with interception proxies. In particular, DHTTP would allow interception proxies to be deployed in arbitrary points in the network and hence enable a wide integration of caches into the Internet fabric.

Many application-level protocols split their traffic between TCP and UDP channels. This includes DNS, which switches from UDP to TCP when responses exceed 512 bytes [10] and RTSP, which uses TCP for control commands and UDP for stream data [11]. Further, FTP servers open TCP connections back to FTP clients for data transfers [10]. We argue that similar approaches are appropriate for Web traffic.

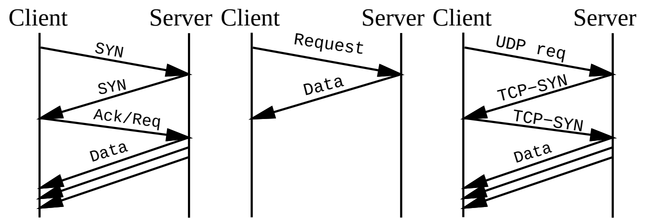
Obviously, DHTTP represents a significant deviation from existing practice. However, it can be introduced incrementally while co-existing with current HTTP in the transitional period (see Section IV-C).

We evaluated the performance of DHTTP by conducting a simulation study as well as by implementing and testing a real DHTTP server, built as a modification of the Apache 1.3.6 Web server [12]. The source code of our DHTTP implementation is available [13].

II. Related Work

Two proposed enhancements to TCP, transactional TCP (T-TCP) [14], [15], and Shared TCP control blocks (S-TCP) [16], address TCP set-up costs at the transport level. Transactional TCP reduces the set-up overhead by caching per-host state from previous connections and allowing the delivery of application data in the first SYN message from a client that has the cached state at the server. T-TCP also allows caching the window size, avoiding the slow-start overhead for consecutive connections from the same client. S-TCP shares slow-start information across concurrent connections between a pair of hosts, helping the connections learn the appropriate window size faster. Still, neither approach relieves the server from the overhead of initializing and maintaining open connections, and neither addresses the violation of the end-to-end semantics by interception caches.

The HTTP-ng initiative [17] proposes to multiplex multiple application-level protocol sessions over the same TCP connection. It allows fragmenting and re-ordering of multiplexed responses and, similar to persistent connections, amortizes TCP connection set-up over multiple fetches. However, it duplicates much functionality of the TCP at the application level, thus introducing unnecessary overhead. For instance, the application level performs its own flow control and its own packet ordering. There is also an extra level of buffering and copying. To be fair, the primary goal of HTTP-ng is not performance but such benefits as “easier evolution of the protocol standard, interface technology that would facilitate Web automation, easier application building, and so on” [18].



(a) Current HTTP. (b) DHTTP over UDP. (c) DHTTP over TCP.

Fig. 1. Message exchange for a Web interaction.

In independent work, Cidon et al. proposed using a hybrid TCP-UDP transport for HTTP traffic [19]. This proposal also splits the HTTP traffic between UDP and TCP. A client sends a UDP request. The server replies by UDP if the response is small; otherwise, the server sends back a special UDP response asking the client to resubmit request over TCP. The client also resubmits the request over TCP if no response arrived within a timeout. Our proposal is similar in its basic premise but differs from Cidon et al. in two major ways. First, unlike Cidon et al., DHTTP servers initiate connections back to the clients, which brings important benefits already mentioned in the Introduction. Second, our mechanism for choosing between TCP and UDP channels explicitly addresses the issue of network congestion. Furthermore, by not prototyping their idea, Cidon et al. could not quantify its affect on server performance.

In response to our posting of an initial draft of this paper on the Internet, we received an unpublished manuscript that also describes splitting Web traffic between TCP and UDP [20]. However, this proposal does not address the network congestion issue, which we found can cause severe performance degradation if traffic is split regardless of network conditions. Another key difference with our protocol is that [20] achieves reliability through client acknowledgments and server re-transmissions. This increases the network overhead for packet acks and server overhead for keeping unacknowledged packets in the buffers and managing per-packet timeouts and retransmissions. Further, by choosing a Perl implementation (with unrealistically low server throughput of under 10 requests per second) and not considering persistent connections and pipelining of existing HTTP in their experimental study, [20] does not make a convincing case for splitting the traffic. We base our experiments on a production Apache server and we include persistent connections with pipelining in our experiments.

Previously, Almeroth et al. proposed to use a UDP multicast for delivery of the most popular Web pages [21]. Using multicast to deliver popular Web pages to proxies has been proposed by Touch [22]. In contrast to these works, we propose to use UDP for much of routine Web traffic.

Analytical models for HTTP performance over TCP and ARDP, an alternative connection-oriented transport protocol built over UDP, are provided and validated in [23]. Unlike our approach, this work does not consider using raw UDP or switching between connection and connectionless transport.

III. DHTTP PROTOCOL

This section describes the protocol and its mechanisms for reliability, flow control, and choosing between TCP and UDP for a response. The next section discusses implications of DHTTP on network caching and security as well as our proposal for deployment.

In DHTTP, both Web clients and servers listen on two ports, a UDP and a TCP. Thus, two communication channels exist between a client and a server - a UDP channel and a TCP channel. The client usually sends its requests over UDP. Only when uploading a large amount of data (e.g., using a PUT request) would the client use TCP. By default, a request below 1460 bytes, the Ethernet maximum transfer unit (MTU), is sent over UDP. Virtually all HTTP requests fall into this category [4][1]. For conceptual cleanness, the client itself initiates the TCP connections to send requests instead of reusing connections initiated by the server for data transfer.

When the server receives the request, it chooses between the UDP and TCP channels for its response. It sends control messages (responses with no data), as well as short (below 1460 bytes, one Ethernet MTU, by default) data messages, over UDP even if there is an open TCP connection to the client. This avoids the overhead of enforcing unnecessary response ordering at the TCP layer. A UDP response is sent in a single UDP packet since our default size threshold practically ensures that the packet will not be fragmented. Dividing a response among several UDP packets would likely allow higher size thresholds and is a promising enhancement for the future. For long data messages (over 1460 bytes by default), the server opens a TCP connection to the client, or re-uses an open one if available.

Figure 1 shows the message exchange of a current HTTP interaction and a DHTTP interaction with the response sent over UDP and TCP. It is important that even when choosing TCP, DHTTP does not introduce any extra round-trip delays compared to the current Web interactions. While it may appear counter-intuitive because in DHTTP, TCP establishment is preceded by an “extra” UDP request, the comparison of Figure 1a and 1c shows that data start arriving at the client after two round-trip times (RTTs) in both cases. In fact, a possible significant (but unexplored in this paper) advantage of DHTTP over current HTTP in this case is that the server may overlap page generation with TCP connection establishment.

Since responses may arrive on different channels and out of order with respect to requests, the client must be able to match requests with responses. Consequently, a client assigns a randomly chosen request ID to each request. The request ID is reflected by the server in the response and allows the client to assign the response to the proper request.

The request ID must be unique only to a given client and only across the outstanding requests that await their responses. We allocate eight bytes for the request ID, sufficient to safely assume no possibility of a collision [25][2].

[1]The cited study is rather old. However, there is some evidence that the overall Web request size remains small despite a growing number of proposals for using HTTP for new applications, which may increase the request size. Considering a 5-month trace from his research department (6 clients), Duchamp found in 1999 the average request size, including all headers, of 189 bytes [24]; in the 1997 trace used for our study [1], 98% of all requests were GET and HEAD, which tend to be small.

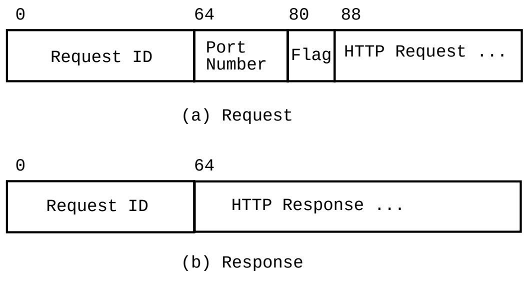


Fig. 2. DHTTP message formats.

The client must also let the server know which ports it listens to on both channels. To save on the overhead, we note that source port number of the channel used by the request is included in the IP headers already. So, the request must include the port number of the other channel only. Consequently, our request message has a port number field, which contains client’s TCP port number if the request is sent over UDP and UDP port number if the request is sent over TCP.

Figure 2 summarizes the DHTTP message formats. In addition to the request ID and port number, the request message includes a byte worth of flags. The only currently used flag is “resend” flag that indicates a duplicate request. Thus, DHTTP adds eleven bytes to every request. The response includes only the request ID, for an eight-byte overhead.

A. Reliability and Non-Idempotent Requests

Given the best-effort nature of the UDP channel, we must provide a reliability mechanism. A straightforward way to provide reliability would be to make clients acknowledge every UDP packet received and servers resend unacknowledged UDP packets. This, however, would increase bandwidth consumption for acknowledgements and server overhead for storing unacknowledged UDP packets and for managing per-packet timeouts and retransmissions. These overheads would be paid whether or not a packet loss occurs.

We believe this approach is never optimal. When packet loss is low, it imposes the unnecessary overheads. When it is high, an implementation would be hard-pressed to compete with highly optimized TCP. So, instead of trying to build reliability into the UDP channel, the DHTTP protocol simply stipulates that a client may resend a UDP request if the response does not arrive for a timeout period, with the resent flag set. A large request timeout (we use 5 and 10 seconds) with a limited number of resends ensures that clients do not overwhelm a server with repeated resends. In principle, clients could use more sophisticated strategies such as smaller initial timeouts followed by exponential backoff.

We leave it to the servers to efficiently deal with resent requests. They may re-generate responses, or cache UDP responses in the buffers so that they can be re-transmitted quickly. However, DHTTP stipulates that a response to a resent request be sent over TCP for flow control (see Section III-B).

[2]In fact, our DHTTP prototype uses only two-byte sequence numbers for request IDs, which we later realized is not adequate given fast improvements in Web proxy performance and some security issues discussed in Section IV-B.

A related issue is support for non-idempotent requests, which should not be re-executed. Examples of such requests include some e-commerce transactions, such as an order to buy or sell stocks. Following its general minimalist approach, DHTTP currently deals with non-idempotent requests by delegating them to TCP transport, instead of providing special support at the application level. In this method, the protocol portion of a URL prescribes the transport protocol to be used by clients. For instance, we can have a convention that, for URLs of the form "dhttpt://<rest-of-URL>", a client must send requests by TCP, while for URLs that start with "dhttp:", it can use UDP. Then, all non-idempotent URLs would be given the "dhttpt:" prefix. Specifying the transport in the protocol portion of a URL is also used in a different context by the RTSP protocol [11].

B. Flow Control

DHTTP servers must avoid flooding a congested network with UDP messages. Instead of implementing its own flow control, DHTTP again leverages TCP by requiring that responses to any resent requests be sent over TCP. So, any time a packet loss occurs, the server switches to TCP with its flow control for this interaction. An HTTP server using MTU discovery [26] sends packets of 1460 bytes over the Internet and has the initial TCP window for data transfer equal to two packets [3]. Thus, DHTTP server could in principle send up to 2920 bytes by UDP without relaxing TCP's flow control. Our current default threshold of 1460 bytes makes DHTTP servers even more conservative than HTTP servers in terms of flow control within one Web download.

One could argue that DHTTP servers may still create traffic bursts by sending a large number of UDP packets belonging to *distinct* parallel downloads. However, short parallel TCP connections will create similar bursts in existing HTTP due to SYN and the first data packets. So, it is only in the case of multiple short downloads to the *same* client reusing a persistent TCP connection in existing HTTP, where DHTTP may be more aggressive. Even in this case, when the fraction of resent requests becomes noticeable (indicating possible congestion), DHTTP servers starts using TCP almost exclusively (see Section III-C). In our experiment over congested Internet, only 6% of responses were sent over UDP. Finally, Feldmann et al. showed that although most Web transfers are short, a majority of the bytes and packets belong to long transfers [27], and DHTTP uses TCP with its native flow control for them.

C. Choosing a Channel

The server must choose between TCP and UDP based on the response size and network conditions. When the network is not congested and packet loss is low, then the best strategy for the server would be to maintain no state for sent responses. This strategy optimizes for the common case of no packet loss, at the expense of having to re-generate the response after a loss does occur.

However, when the network is congested, this strategy is extremely poor. Not only do the UDP responses have to be re-regenerated and re-transmitted often, but even TCP responses may arrive at clients so slowly that clients send duplicate requests for them. The result is that the server sends many duplicate responses, further aggravating network congestion. The same situation may occur with compute-intensive responses which may take a long time to reach the client.

[3]The initial window size is 1 but most implementations increase it after receiving the TCP SYN-ACK packet.

To address this issue, our server maintains a "fresh requests counter", incremented any time the server sends a response by UDP to a request with unset resend flag, and a "resent requests counter", which counts the number of resent requests received.

Our algorithm for choosing a channel uses a *loss threshold* parameter, L, (currently 1%) and a *size threshold* parameter s (1460 bytes by default). All responses exceeding the size threshold as well as those in reply to resent requests, are sent over TCP. The choice for the remaining responses depends on the ratio of resent request counter to fresh request counter. If this ratio is below L, these responses use UDP. The ratio above L indicates high packet loss and would suggest sending all responses by TCP. However, the server must still send a small number of responses over UDP to monitor the loss rate, since losses in the TCP channel are masked by the TCP layer. Therefore, we chose rather arbitrarily to send $1 - L$ fraction (or 99%) of small responses over TCP and the remaining small responses over UDP in the high loss condition.

There is still a race condition mentioned earlier, where a client may time out and resend a request before the TCP response to this request arrives. To address this race condition, our server maintains a circular buffer of global request IDs (which is a combination of the client IP address and request ID from a request) that have been responded to by TCP. The buffer has room for 10000 global request IDs. When a resent request arrives, the master process ignores it if it is found in the buffer, since the response was already sent by TCP that has reliable delivery.

A potential limitation of the above algorithm is that the server maintains aggregate packet loss statistics. While the aggregate statistics reflect well the congestion of the server link to the Internet, congestion of network paths to individual clients may vary. Thus, enough clients with congested links can make the server use TCP even for clients with uncongested links. Conversely, the server may use UDP for an occasional congested client if its connection to the rest of the clients is uncongested. The investigation of finer grain algorithms remains a future work. At this point, we only note that if a UDP response to the congested client is lost, the client will resend its request with the *resent* flag set, forcing the server to use TCP for this interaction.

Choosing a size threshold presents another interesting tradeoff. A large value will reduce the number of TCP connections by sending more responses over UDP; however, if it exceeds one MTU (1460 bytes), some responses in the current version of DHTTP will be fragmented. Fragmentation degrades router performance [28] and entails resending the entire response upon a loss of any fragment. Thus, in a high loss environment such as Internet, s should be limited to one MTU. Future versions of DHTTP will be able to use higher values of s by sending a large response over several UDP packets, which would avoid fragmentation and allow clients to issue *range requests* [7] to obtain just the missing portions of the response in the aftermath of a packet loss.

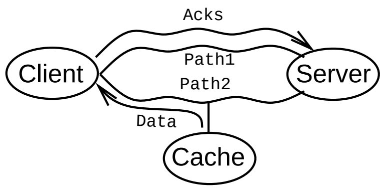


Fig. 3. Problematic deployment of interception cache.

IV. Implications of DHTTP

A. DHTTP and Interception Caches

Interception caching has been an enticing idea for ISPs since it allows them to cache Web content in the network transparently to clients. With interception caching, routers or switches on the request path to the server divert the request to a proxy cache, which accepts the connection and send the response as if it were the origin server. To impersonate the origin server, the cache uses the IP address of the origin server as the source IP address of the response packets. Since this breaks the end-to-end principle of the Internet, interception caching has raised much controversy[4]. The main concerns are the possibility that an interception cache may disrupt TCP connections and that it deceives clients into assuming they interact with end servers, so that the clients may neglect adding appropriate cache control headers into their requests.

Connections can be disrupted when different packets from the client choose different paths to the server. For instance, in Figure 3, assume that half way through the download of a page from the cache, TCP acknowledgements from the client will choose the upper path and start arriving at the origin server, which will discard them since it does not recognize the connection. In the meantime, the cache will not receive any acknowledgements and will eventually timeout the connection.

These problems may occur even without route changes, e.g., when OSPF routers forward packets using round-robin load balancing over multiple shortest path routes. In fact, with route load balancing, an inappropriate placement of interception cache in the network may disrupt *every* Web interaction going through the ISP. In any case, most ISPs are not willing to add extra connection failures, however few. Thus, they limit deployment of interception caches to only network points traversed by *all* packets from clients. In particular, they typically avoid interception caches in transit backbones and turn off caching for requests arriving from another ISP or from clients known to obtain Internet connectivity from multiple ISPs.

DHTTP retains the end-to-end semantics even with interception caching. A DHTTP interception cache will intercept only requests sent over UDP and pass through any requests using TCP. As already mentioned, these requests will be either data uploads to servers or requests for non-idempotent resources, the kinds of requests that usually cannot benefit from caches anyway. So, restricting interception caches to UDP requests will not reduce cache effectiveness.

When a DHTTP cache intercept a UDP request, it will either respond by a UDP packet, or establish a connection back to the client. In either case, it will use its true source IP address (refer to Section IV-B for security considerations). At the transport layer, no IP impersonation occurs and all packets will arrive at each end regardless of routing path properties. Thus, DHTTP would enable a wide integration of interception caches into the Internet fabric, without any consideration of routing policies.

Further, the client is aware it speaks with the cache and not the end server since the response comes from a different IP address. A configuration option can thus be easily added to clients that would allow them to bypass all interception caches or selectively allow or exclude interception caches on certain subnets or for requests to certain Web sites.

B. Firewalls and Security

DHTTP imposes similar requirements on firewalls as many other protocols. In particular, the firewall must open a limited-time “hole” to the client UDP and TCP ports specified in a DHTTP request, so that the DHTTP response can get through. Modern firewalls, such as widely used FireWall-1 [29], provide this functionality for other protocols and are capable for maintaining the necessary state from previous requests. For example, for RTSP protocol, FireWall-1 opens a temporary hole for incoming UDP packets to the client’s port specified in the client’s SETUP request. To support FTP in active mode, FireWall-1 opens a similar hole for incoming TCP connections to the port specified in the client’s PORT command.

DHTTP shares with active FTP the vulnerability to an attack through a malicious applet described in [30]. In this attack, a malicious site entices the client to visit its Web page, which contains an applet that generates an FTP request with the sole goal of opening the hole to incoming TCP connections to a vital port, such as port 23 used by telnet, giving the attacker an opportunity to try a login password. By the same token, the remedy described in [30] and implemented in Firewall-1, which is to prohibit low port numbers used by telnet and other entry points, also applies to DHTTP.

Further, if interception proxies are allowed to use their true IP addresses, the firewall must let packets with an arbitrary source IP address through the hole to the client ports. This is a deviation from existing firewalls that only allow incoming packets from the IP address matching the destination IP address of the request packet that opened the hole. Since learning this IP address requires an attacker to have the ability to intercept request packets, removing this requirement may seem as if lowering the bar for the attacker. A closer analysis, however, shows that even with DHTTP, the attacker would still have to intercept request packets. First, the attacker still has to learn the correct client port number to get through the firewall, and it has only short time to guess the port before the hole closes. To complicate guessing port numbers, a client using DHTTP in the clear can frequently change its port numbers, every time choosing a different random port number[5]. Second, even if the imposter guessed the port number in time and passed through the firewall, its packets would be discarded by the client because they will not match a valid request ID number (the likelihood of guessing an 8-byte

[4]See the email discussion in http://www.wrec.org/archive/, especially the threads “Recommendation against publication of draft-cerpa-necp-02.txt” and “Interception proxies” in April, 2000.

[5]While many OS kernels seem to allocate smallest available port numbers, changing it to random numbers is straightforward.

random request ID can be safely dismissed [25]). Learning the correct request ID would again require intercepting the request. A potential denial of service attack on the client machine by flooding it with packets containing wrong request IDs could afflict limited damage because the guessed port will remain valid only for the duration of the current hole. Current Web servers are much easier (because they have a permanent hole for incoming connections to a well-known port) and more enticing targets for these attacks. The attacker can also attempt a SYN attack against a DHTTP server by sending DHTTP requests for large objects to arbitrary other DHTTP servers with the attacked server's IP address as the source address. A simple defense against such an attack is to allocate non-overlapping port number ranges to DHTTP servers and clients, allowing servers to discard SYN packets to client ports.

We stress that both DHTTP and existing HTTP are vulnerable to an imposter with a capability to intercept and examine request packets. In particular, such imposter can substitute the legitimate content with its own. In existing HTTP, it would do so by learning the intended Web server IP address, and in DHTTP by learning the client port number and request ID. Only an encrypted version of the protocol, be it HTTP or DHTTP, can protect against such an attack.

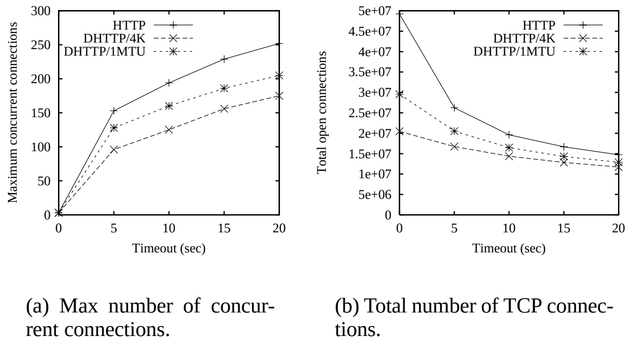
C. *Incremental Deployment*

One cannot realistically assume that the world will switch to DHTTP at once. A simple incremental path to deployment is as follows. DHTTP clients and servers must be able to also use HTTP in the transitional period. Further, in the transitional period, DHTTP clients should use existing HTTP for any requests sent by TCP (such as large or non-idempotent requests), even to presumably a DHTTP server, so that legacy HTTP servers never receive DHTTP requests on a TCP channel.

A URL naming convention would be established between DHTTP servers and clients, so that the latter will recognize URLs hosted by DHTTP servers *in most cases*. For example, a convention can be that DHTTP URLs have the form of "http://host-name/_dhttp_/remainder-of-URL". Legacy HTTP clients will issue a normal HTTP request for these URLs while DHTTP clients will use DHTTP for them, unless it is a request using TCP. There is still a remote possibility that a legacy HTTP server uses a URL of the same format, in which case a DHTTP client may issue a DHTTP request over UDP to this server (recall that any request over TCP is sent using legacy HTTP format). Unless this server also happens to use this UDP port for an unrelated application and has no barrier to block an unexpected UDP message, the client will get back an ICMP "destination unreachable" message and immediately resend the request using HTTP. In theory, an HTTP server that has a URL with DHTTP format *and* and uses the DHTTP UDP port for an unrelated application *and* does not block outside messages to this port could still receive an unexpected DHTTP request. However, the combination of all these conditions is so unlikely that we consider it impossible for all practical purposes.

V. Performance Analysis

We performed a three-pronged performance study. First, we conducted a trace-driven simulation to study the number and utilization of TCP connections that the server experiences under HTTP and DHTTP. Second, we benchmarked the Apache HTTP server and our DHTTP server with clients on the same LAN, to compare their peak performance and scalability. Finally, we tested both servers in a WAN environment with a congested Internet connection.



(a) Max number of concurrent connections.

(b) Total number of TCP connections.

Fig. 4. Simulation results with three connections per client.

A. *Simulation*

Our simulation study uses the access log from the EasyWWW Web server, which provides AT&T's low-end hosting services. The trace has the duration of three months and contains over 100 million accesses with the average response size of 13K. Each log record contains the client address, the URL requested, response code, size of the response and the timestamp. We add an appropriate number of bytes to the size of the response for response and IP headers (the actual number depends on the response code).

Our simulation assumes that the time to generate a response at the server is negligible compared to the value of persistent connection timeout. Thus, we measure the time between request arrivals for persistent connection timeouts. In reality, the server measures the time between the completion of one response generation and the arrival of the next request. However, the information needed to compute this time interval is not available in server logs.

The simulation processes log records in timestamp order. In the HTTP case, for each record, if the number of connections from the client reached the maximum, the oldest connection is reused. Otherwise, a new connection is opened. After the access at time t, the connection is closed at time $t+TIMEOUT$ unless it is reused before that.

For the DHTTP case, the response is sent by UDP if the size of the response is less than a threshold value s. Otherwise, it is sent by TCP, reusing existing connections if available. We use two threshold values in our simulation - 4K and 1460 bytes. The former is an optimistic value based on multiple studies showing that many Web responses are smaller than 4K bytes; the latter is a conservative value that fits into one Ethernet MTU.

We are concerned with three performance measures: the maximum number of simultaneous connections observed at the server during the simulated period, the total number of connections used by the entire trace, and average connection utilization. The first metric indicates the scalability requirements of

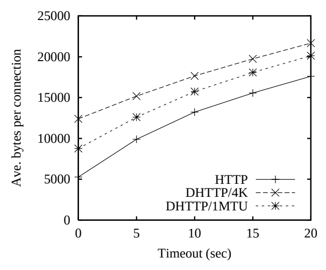


Fig. 5. Simulation results with three parallel connections per client: connection utilization.

the server at peak demand [6]. The second metric shows how many requests paid an overhead of TCP connection establishment during the trace period. Connection utilization measures the amount of data sent over a TCP connection. TCP is optimized for large data transfers, which allow it to learn available bandwidth and amortize start-up costs. Overall, we would like low numbers of simultaneous and total connections and high connection utilization.

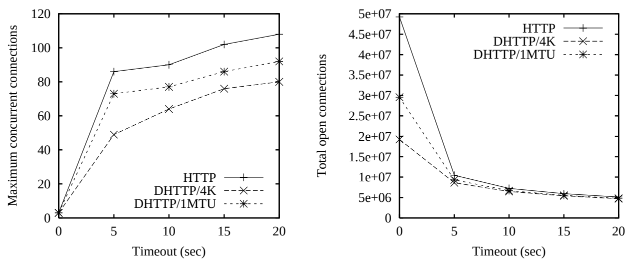
Figures 4 and 5 show the simulation results when clients can open up to three parallel connections to the server. For Apache's default connection timeout of 15 seconds, our approach reduces the maximum number of simultaneous connections by a third for the 4KB threshold and by 20% for the 1460 bytes threshold. The difference can be reduced by decreasing the timeout value, which has in fact been recommended for servers under high load [9], [31]. But that increases dramatically the total number of connections (Figure 4b), meaning that more requests must pay the overhead of connection establishment and slow-start. Thus, DHTTP has a significant advantage over HTTP over the entire range of timeout values, with most benefits due to the number of simultaneous connections at high end of the range and the total number of connections at low end.

In the DHTTP case, the fraction of responses sent over UDP was over 58% for the 4K threshold and over 40% for the 1460 bytes threshold. Since TCP is used only for large responses, DHTTP exhibits better utilization of TCP connections. As shown in Figure 5, the average number of bytes sent over one connection can be over two times higher in DHTTP than in HTTP, for 4K size threshold and short timeout values.

Figures 6 and 7 depict the results for the case where all clients only use one connection. Figure 6a shows that while the maximum number of simultaneous connections decreased for both HTTP and DHTTP, the relative difference between the two can still be significant, reaching almost the factor of two for 5 second timeout and 4K size threshold. Further, Figure 6b shows that the total number of connections opened by HTTP drops only for high timeout values, where it was already low. Thus, DHTTP retains a significant overall performance advantage over the entire range even under an unrealistically conservative assumption that all clients open only one connection at a time to any server [7]. Figure 7 shows that DHTTP's advantage in connection utilization, while still noticeable, is much smaller than in the case of three connections per client, except for short timeouts where DHTTP's advantage remains practically the same.

[6]The maximum number of concurrent connections a server can support is one of main parameters provided by server vendors and tested by benchmarks.

[7]The assumption is unrealistic because there are valid performance reasons, such as the head-of-line delay in pipelining mentioned in Introduction, why HTTP 1.1 clients open more than one connection to a server.



(a) Max number of concurrent connections.

(b) Total number of TCP connections.

Fig. 6. Simulation results with a single connection per client.

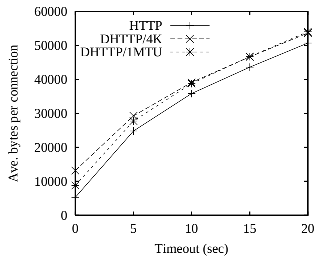


Fig. 7. Simulation results with a single connection per client: connection utilization.

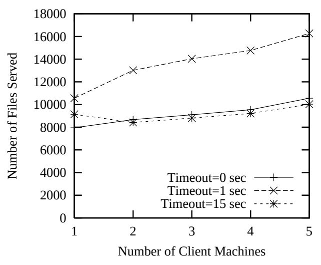
B. Prototype Testing

We tested the original Apache server and our modified version that speaks DHTTP (referred to as DHTTP server) using the SURGE workload generator [32]. We found it to be the most appropriate workload generator for our purpose because it models the client idle times, allowing TCP connections to time out in a realistic manner. Beside idle times, SURGE tries to match empirical distributions of document sizes on the server, request sizes, document popularity, embedded object references, and temporal locality of reference. Also important for us is that SURGE imitates HTTP1.1 clients, utilizing persistent connections and pipelining.

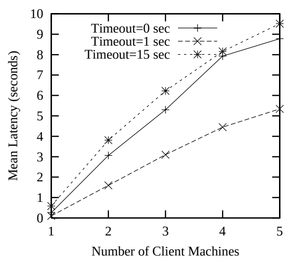
SURGE forks several processes, each creating multiple threads that simulate the behavior of individual users. We used current SURGE to test Apache. We also modified SURGE to use DHTTP and used this version to test our DHTTP server. All experiments ran for two minutes.

We chose Apache as the basis for our implementation because it is by far the most popular Web server. Recently, more efficient event-based servers have been built (e.g., [33], [34]). We believe our conclusions are not affected by the process-based implementation of Apache because our DHTTP server is also process-based and inherits the same overheads. If anything, DHTTP is hurt more by the process-based architecture because it has higher throughput and therefore more context switches.

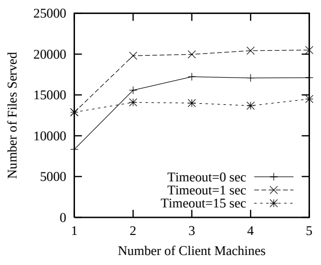
To maximize the throughput of current Apache, we configured it to the maximum number of concurrent clients of 200



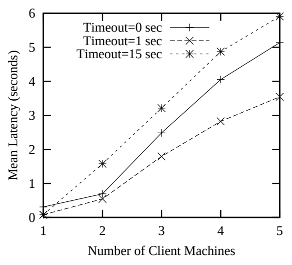
(a) Throughput with three connections per client.



(b) Latency with three connections per client.

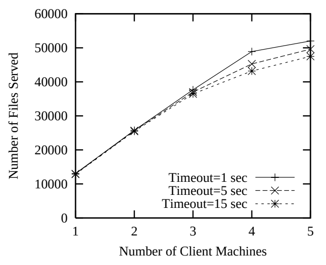


(c) Throughput with one connection per client.

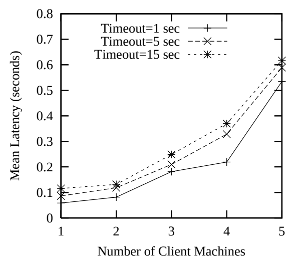


(d) Latency with one connection per client.

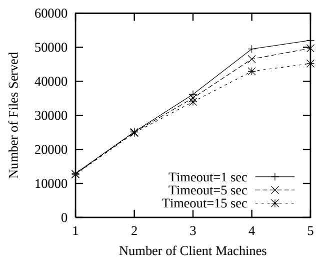
Fig. 8. Apache performance (bottleneck at the server).



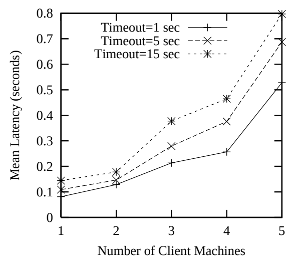
(a) Throughput with three connection per client.



(b) Latency with three connection per client.



(c) Throughput with one connection per client.



(d) Latency with one connection per client.

Fig. 9. DHTTP server performance (bottleneck at the server).

(the default is 150). Increasing the limit further did not lead to any additional performance gains since the server could not fork any more worker processes anyway. Since each worker process has at most one TCP connection and because standard Linux limits the total number of TCP connections that are in the process of being established (the server sent the SYN-ACK and is waiting for ACK from the client to complete the handshake) or are waiting to be accepted to 128, the total number of simultaneous connections at the Apache server can be at most 328. To factor out the effect of this limit, we artificially limited the number of concurrent TCP connections in the DHTTP server to the same value. Since both servers share the same connection limit, setting it to another value should not significantly affect performance trends. We note, however, that in reality DHTTP servers do not have a bottleneck process that accepts all TCP connections and therefore the number of TCP connections in DHTTP is only limited by the maximum number of open sockets. All experiments used 1460 bytes for the site threshold and 1% for the loss threshold values.

B.1 Bottleneck at Server

To test the servers for peak performance, we conducted a study in a fast LAN environment, to ensure that the bottleneck is at the server. Our LAN setup included a server on a Pentium III 500MHz PC running Linux and a variable number of SGI workstations running SURGE clients, all connected by a 100Mbps LAN. We ran 4 processes on each client machine, with 50 threads per process for the total of up to 1000 user equivalents on 5 machines.

Figure 8 shows the throughput (reported as the number of requests served during the two-minute experiment) and latency of Apache, when the maximum number of parallel TCP connections to a client is set to three and one. Figure 9 shows the same characteristics for the DHTTP server. We see a dramatic performance advantage of DHTTP - an order of magnitude lower latency and several times higher throughput. The DHTTP server also scales better and is much less sensitive to different persistent connection timeout values and the number of concurrent TCP connections to a client.

Interestingly, Apache performance is the worst for the default value of persistent connection timeout, 15 seconds, except for the extremely light load. We found the best timeout value to be 1 second, and also that the performance improves if clients never open multiple parallel connections to the server. A similar observation was made in [9].

We found the reason for this is that Apache never forcefully closes idle TCP connections until they timeout, even if other connections are waiting in the accept queue. The same reason explains the negative effect of multiple concurrent connections to a client - it increases the number of connections at the server, with more idle connections blocking incoming connections in the queue.

To factor out this implementation artifact, we modified Apache to check the TCP accept queue every second during periods when it has idle connections. Whenever the accept queue is not empty it closes some idle connections so that pending TCP connections can be served. The resulting behavior is that the server uses high timeout when it has spare connections and switches to a sub-second timeout otherwise. Recent proposals for dynamic adjustments of the timeout value based on the load

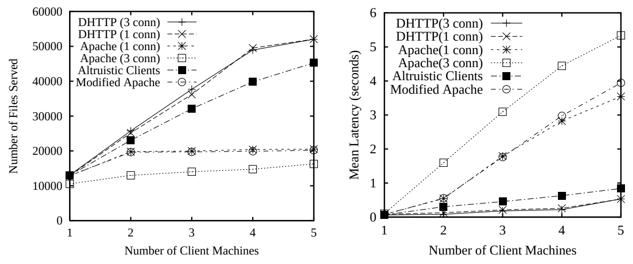


Fig. 10. Comparison of Apache and DHTTP servers: throughput (left) and latency (right).

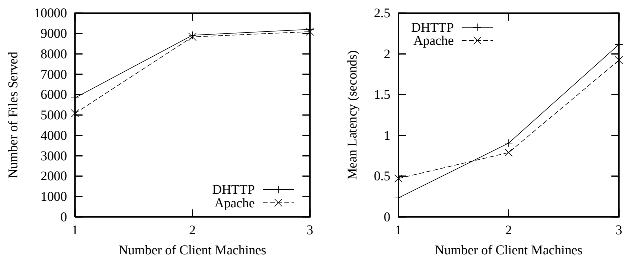


Fig. 11. Apache and DHTTP server performance under network congestion: throughput (left) and latency (right).

and request history indirectly address the same issue [31].

Further, SURGE clients never close connections, hoping to reuse them for future requests. To test a more altruistic client behavior, we modified SURGE to follow a recently proposed *early close* policy [9] where clients close their connections after getting all the files in one HTML page, thereby reducing the number of connections that servers must handle.

Figure 10 shows the performance of Apache that was modified as discussed (the "Modified Apache" curve) and also the performance of the current Apache with clients using early close policy (the "Altruistic clients" curve). For comparison, it also duplicates the curves for DHTTP and Apache from Figures 8 and 9, for the best persistent connection timeout value (1 second). The modified Apache has the timeout value of 15 seconds.

The figure shows that the performance of the modified Apache is very close to the the current Apache with TCP timeout of 1 second. Since for a non-overloaded server a longer timeout would result in lower latency to the clients, we believe this modification is worthwhile. Altruistic clients result in a huge performance improvement for Apache. Still, DHTTP achieves around 20% better throughput and twice as low latency than Apache with altruistic clients.

B.2 Bottleneck in the Network

To test the behavior of DHTTP under network congestion, we conducted an experiment over the Internet. For this study, we used up to three client machines, located at University of Washington, Duke University, and New York University. The server was at AT&T Labs in Florham Park, connected to the Internet by a fractional T3 line with a bandwidth of up to 9Mbps. We ran two SURGE processes with 50 threads on each client machine, to observe the DHTTP behavior on the Internet with the network below, near, and at saturation[8]. The persistent connection timeout for both servers is 5 seconds. Since the TCP accept queue is no longer the bottleneck, the connection timeout value did not affect performance. For the same reason, current Apache, altruistic clients and modified Apache all showed very similar performance. We describe here the performance of current Apache.

[8]From Figure 10, four SURGE processes generate roughly 10000 requests in two minutes, which translates into over 80 requests per second. With the average file size of roughly 10K, this generates around 8Mbps traffic. So, two client machines in our WAN experiment should saturate the connection, three would put it far over the edge and one machine by itself should leave the network very close to but below saturation most of the time, unless the path from the client to the server includes lower-bandwidth links or there is competing traffic.

TABLE I
EFFECTIVENESS OF CONGESTION DETECTION IN DHTTP SERVER.

Client machines	fraction of UDP responses	resend/fresh ratio	ignored resent requests
1	18.9%	0.7%	0
2	8.3%	1.2%	4.9%
3	6.1%	4.9%	35%

Recalling that the UDP channel is not appropriate for these conditions, our goal is to see how successful the server is in monitoring the congestion and switching to TCP. Figure 11 shows throughput and latency of Apache and DHTTP. DHTTP outperforms Apache with one client machine - it has 15% higher throughput and half the latency of Apache. On a saturated network, Apache has slightly better latency (by up to 13%) and virtually identical throughput. The increased latency of DHTTP is probably due to fact that when a UDP request is lost, the client waits for the fixed timeout before resending it.

Overall, these results indicate that the DHTTP server successfully detects the network congestion and switches to TCP. Table I provides some insight into the effectiveness of its congestion detection mechanism. We can see that the the fraction of responses sent over UDP channel indeed drops significantly as the network congestion grows worse. At the same time, the number of resent requests remains low (recall that the resend/fresh ratio numbers in the third column are relative to the decreasing number of UDP responses). Also, with three machines, 35% of resent requests are ignored by the server. This means the server already sent the responses by TCP, but the congestion is so high that the requests timeout at the clients before the responses arrive.

VI. CONCLUSIONS

This paper describes and motivates a new protocol for Web traffic, DHTTP. The protocol splits the traffic between UDP and TCP channels based on the size of responses and network conditions. When TCP is preferable, DHTTP opens connections from Web servers to clients rather than the other way around.

With existing HTTP, busy servers face a dilemma of *either* limiting the number of Web accesses that benefit from persistent connections *or* having to deal with a large number of simultaneous open connections. By performing short downloads over UDP, DHTTP reduces *both* the number of TCP connection set-ups *and* the number of open TCP connections at the server. At the same time, the TCP connections that DHTTP does open transfer larger objects, increasing connection utilization. Also, by opening TCP connections back to the clients, the server no longer has a bottleneck process that receives all TCP connection requests from the clients.

Our performance analysis shows that when the network is not congested, DHTTP significantly improves the performance of Web servers, reduces user latency, and increases utilization of remaining TCP connections, improving their ability to discover available bandwidth. At the same time, we demonstrate that the DHTTP server successfully detects network congestion and uses TCP for almost all traffic under these conditions.

From the functionality standpoint, DHTTP retains the end-to-end Internet semantics in the presence of interception Web proxies, allowing their unconstrained deployment throughout the Internet without the possibility of disrupting connections.

As a future work, we would like to finesse the channel selection algorithm. First, it would be interesting to consider finer-grain statistics on resent requests and factor the client subnet into channel selection. Second, we would like to explore changing the UDP size threshold dynamically based on the observed packet loss. When the loss rate is low, we could double the size threshold to two Ethernet MTUs without relaxing TCP's flow control, for further performance gains. This would require sending a UDP response in several packets to avoid packet fragmentation.

Finally, since the server in DHTTP can decide how many connections to open to a client, it would be interesting to capitalize on this capability to improve performance. For example, if the server has few open connections overall, it may decide to open many TCP connections to the client to parallelize sending multiple embedded objects If the server already has a large number of connections, it may send these objects sequentially over the same connection, or even use individual short-lived connections in a succession.

The source code for our DHTTP server implementation, as well as SURGE workload generator modified to use DHTTP, is available [13]. Beyond benchmarking DHTTP servers, modified SURGE can also serve as an example of a DHTTP client implementation.

ACKNOWLEDGMENTS

We would like to thank Paul Barford for making SURGE available and for his help with our SURGE questions. We are indebted to Jennifer Rexford for numerous insightful comments on earlier drafts of the paper. We are grateful to Balachander Krishnamurthy for the traces used in our simulation study, and to Ed Lazowska, Erik Lundsberg, and Amin Vahdat for access to the SGI machines used in our Internet experiments. Thanks also go to Steve Bellovin, Matthew Roughan, and Avi Rubin for their help with security and firewall issues, and to Fred Douglis, Larry Masinter, Mikhail Mikhailov, and Graig Wills for useful discussions, comments, and suggestions.

REFERENCES

[1] A. Feldmann, R. Cáceres, F. Douglis, G. Glass, and M. Rabinovich, "Performance of web proxy caching in heterogeneous bandwidth environments," in *INFOCOM-99*, 1999.

[2] M. Arlitt, R. Friedrich, and T. Jin, "Workload characterization of a Web proxy in a cable modem environment," Tech. Rep. HPL-1999-48, HP Labs, Apr. 1999.

[3] G. Abdulla, *Analysis and Modeling of World Wide Web Traffic. PhD Dissertation*. Virginia Polytechnic Institute and State University, 1998.

[4] B. Mah, "An empirical model of HTTP network traffic," in *Proceedings of the INFOCOM'97*, 1997.

[5] P. Barford, A. Bestavros, A. Bradley, and M. Crovella, "Changes in Web client access patterns," *Wolrd Wide Web*, Special issue on Characterization and Performance Evaluation 1999.

[6] V. N. Padmanabhan and J. C. Mogul, "Improving http latency," *Computer Networks and ISDN Systems*, vol. 28, pp. 25–35, Dec. 1995.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616: Hypertext transfer protocol — HTTP/1.1." ftp://ftp.internic.net/rfc/rfc2616.txt, June 1999.

[8] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, "Network performance effects of HTTP/1.1, CSS1, and PNG," in *Proceedings of ACM SIGCOMM'97 Conference*, Sept. 1997.

[9] P. Barford and M. Crovella, "A performance evaluation of hyper text transfer protocols," in *Proceedings of SIGMETRICS-99 Conference*, pp. 188–197, 1999.

[10] R. Stevens, *TCP/IP Illustrated, Volume 1*. Reading, MA: Addison-Wesley, 1994.

[11] H. Schulzrinne, A. Rao, and R. Lanphier, "RFC 2326: Real Time Streaming Protocol (RTSP)." ftp://ftp.internic.net/rfc/rfc2326.txt, Apr. 1998.

[12] "Apache server project." `http://www.apache.org/httpd.html`.

[13] "DHTTP." http://www.research.att.com/ misha/dhttp/abstract.html.

[14] R. Braden, "Extending tcp for transactions - concepts," November 1992. RFC-1379.

[15] R. Stevens, *TCP/IP Illustrated, Volume 3*. Reading, MA: Addison-Wesley, 1996.

[16] J. Touch, "Tcp control block interdependence," April 1997. RFC-2140.

[17] M. Spreitzer and W. Janssen, "HTTP "Next Generation"," in *Proceedings of the 9th Int. World Wide Web Conference*, May 2000.

[18] "Short- and long-term goals for the http-ng project." http://www.w3.org/Protocols/HTTP-NG/, March 1998. W3C Working Draft.

[19] I. Cidon, R. Rom, A. Gupta, and C. Schuba, "Hybrid TCP-UDP transport for Web traffic," in *IEEE Int'l Performance, Computing and Communications Conference*, 1999.

[20] B. Brown and J. Pasquale, "U-HTTP: a high performance UDP-based HTTP." Unpublished manuscript.

[21] K. C. Almeroth, M. H. Ammar, and Z. Fei, "Scalable delivery of web pages using cyclic best effort (udp) multicast," in *INFOCOM*, 1998.

[22] J. Touch, "The lsam proxy cache - a multicast distributed virtual cache," in *3rd Int. WWW Caching Workshop*, 1998.

[23] J. Heidemann, K. Obraczka, and J. Touch, "Modeling the performance of HTTP over several transport protocols," *IEEE – ACM Transactions on Networking*, vol. 5, pp. 616–630, Oct. 1997.

[24] D. Duchamp, "Personal communication," 2000.

[25] A. Menezes, P. V. Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997.

[26] J. Mogul and S. Deering, "Path MTU discovery, RFC 1191." http://dynamo.bns.att.com/rfc/rfcdir/rfc1191.txt, 1990.

[27] A. Feldmann, J. Rexford, and R. Cáceres, "Efficient policies for carrying Web traffic over flow-switched networks," *IEEE/ACM Transactions on Networking*, vol. 6, pp. 673–685, Dec. 1998.

[28] C. A. Kent and J. C. Mogul, "Fragmentation considered harmful," in *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*, pp. 390–401, Aug. 1988.

[29] "Check point firewall-1." Check Point Software Technologies. White Paper. http://www.checkpoint.com/products/firewall-1/index.html.

[30] D. M. Martin, S. Rajagopalan, and A. D. Rubin, "Blocking Java applets at the Firewall," in *Internet Society Symposium on Network and Distributed Systems Security*, 1997.

[31] E. Cohen, H. Kaplan, and J. D. Oldham, "Managing TCP connections under persistent HTTP," in *Proceedings of the 8th International WWW Conference*, (Toronto, Canada), May 1999.

[32] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of SIGMETRICS-98 Conference*, pp. 151–160, 1998.

[33] "THTTPD - tiny/turbo/throttling HTTP server." `http://www.acme.com/software/hthhpd/`.

[34] "Zeus web server." `http://www.zeus.co.uk/products/zeus3/`.