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

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

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

Abstract—

Today's HTTP carries Web interactions over client-initiated TCP connections. An important implication of using this transport method is that *interception caches* in the network violate the end-to-end principle of the Internet, which severely limits deployment options of these caches. Furthermore, while an increasing number of Web interactions are short, indeed frequently carrying only control information and no data, TCP is often inefficient for short interactions.

We propose a new transfer protocol for the Web, 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. Through server-initiated connections, DHTTP upholds the Internet end-to-end principle in the presence of interception caches, thereby allowing unrestricted caching within backbones. Moreover, 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.

I. Introduction

This paper addresses two important issues in the current HTTP protocol: the violation of the end-to-end principle of the Internet by interception caches and performance implications of using client-initiated TCP as the transport protocol.

Interception caches [31] intercept client requests on their path to origin servers and respond to clients on servers' behalf. Interception caching is attractive to Internet Service Providers (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 principle* of the Internet [33]. One particular implication of this violation is that an interception cache can disrupt a Web intercation if some client packets bypass the interception cache on their way to the server (see Section IV-A). As a result, interception caches can only be deployed safely if there is a point in the network that is guaranteed to see all packets from a given client. This implies significant deployment limitations for interception caches. In general, because of its violation of the end-to-end principle, interception caching gave rise to much controversy in the Internet community.[1]

With regard to performance, 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 [15], 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 [4]. Median response sizes of 1.5-3KB were also reported in numerous earlier studies, e.g., [1], [23], [5]. Such behavior is not always served well by TCP because of high overhead of TCP connection establishment and because of the strain connection maintenance places on the servers.

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* [17]. 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 [28]. 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 [7]. Persistent connections increase the number of open connections at the server, which can have a significant negative effect on server throughput. In fact, a study of 500 top Web sites showed that a quarter of them did not support persistent connections in the Fall of 2000 and that the number of such sites dropped only from 29 to 25% during the preceding 18 months of study [21]. We speculate that server throughput concerns play an important role in this phenomenon.

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 [7] and Section VI-B).

To limit the number of open connections, servers close connections that remain idle for a *persistent connection timeout* pe-

A preliminary version of this paper appeared at INFOCOM'01.

[1] 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.

riod. Busy sites often use short connection timeouts, which reduces the number of open connections but at the same time increases the number of requests that pay the connection set-up costs (see Section VI-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 [37]).

The Dual-Transport HTTP protocol (DHTTP) described in this paper incorporates two main ideas. First, it 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 open connections at the server *and* the number of TCP connection set-ups. In this way, DHTTP improves client latency, since fewer Web interactions wait for the connection set-ups, and increases server capacity by reducing the number of open connections servers must manage. 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.

The second idea behind DHTTP is that, 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-C), this role reversal brings major benefits. Most importantly, it allows interception caches to use their true IP addresses in their communication with clients. Thus, DHTTP retains the end-to-end principle 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. Furthermore, as we will see in Section III, server-initiated TCP 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, even though TCP set-up is preceded by an "extra" UDP request message from the client. (Of course, when the server uses UDP, the number of message round-trips decreases.) Finally, it removes a bottleneck process at the server that accepts all TCP connections.

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 [37] and RTSP, which uses TCP for control commands and UDP for stream data [34]. Further, in the active mode of the FTP protocol, FTP servers open TCP connections back to FTP clients for data transfers [37]. 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-E). 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 [3]. The source code of our DHTTP implementation is available [14].

II. Related Work

Several performance enhancements to TCP address TCP set-up costs. The TCP Fast Start technique [29] allows caching the congestion window size, avoiding the slow-start overhead for consecutive connections from the same client. The shared TCP control blocks optimization (S-TCP) [40] shares slow-start information across concurrent connections between a pair of hosts, helping the connections learn the appropriate window size faster. The Transactional TCP (T/TCP) extension piggybacks data on the handshake control segments, thus avoiding the latency penalty for performing the handshake prior to page download. Still, none of these approaches relieves the server from the overhead of initializing and maintaining open connections, nor do they address the violation of the end-to-end principle by interception caches. We provide a detailed comparison of DHTTP with T/TCP later in Section VI-B.3.

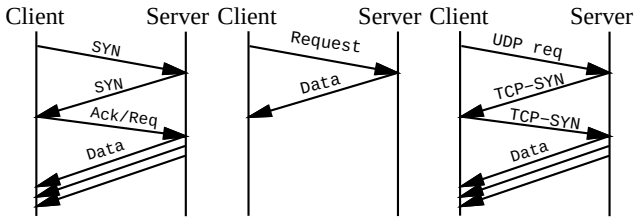
The TPOT approach by Rodriguez et al. [32] addresses the end-to-end issue of interception caches at the TCP layer, by switching to the true IP address of the interception cache during TCP hand-shake. Unlike our approach, TPOT requires a change to the TCP stack and does not address its performance implications.

The HTTP-ng initiative [35] 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" [19].

The proposal of this paper is different from all of these approaches in that it asserts that a TCP pipe is not always an appropriate transport for Web traffic. Consequently, our proposal allows the parties to choose and go back and forth between the UDP and TCP channels.

In independent work, Cidon et al. proposed using a hybrid TCP-UDP transport for HTTP traffic [12]. 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 allows interception caches to not violate the Internet's end-to-end principle and brings other 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.

A proposal to split Web traffic between TCP and UDP was



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

Fig. 1. Message exchange for a Web interaction.

also described by Brown in his Master's Thesis [10]. 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 difference with our protocol is that [10] 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. Also, while proposing that servers open TCP connections to clients, [10] does not make a connection with unconstrained deployment of interception caches that this arrangement allows, which is a key insight of the current paper. Finally, 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, [10] 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 [2]. Using multicast to deliver popular Web pages to proxies has been proposed by Touch [41]. 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 protocol built over UDP, are provided and validated in [18]. Unlike our approach, this work does not consider using raw UDP or switching between connection and connectionless transport.

III. DHTTP PROTOCOL

In DHTTP, both Web clients and servers listen on two ports, a UDP and a TCP, except that servers use well-known ports (or ports specified in the URLs) and clients use so-called ephemeral ports, that is, ports selected anew for a given download. 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. Even if an open TCP connection to the server is available, the client uses UDP because it cannot be sure the server still keeps the TCP connection on its end. Only when uploading a large amount of data (e.g., using a PUT request) and in other special cases discussed later would the client use TCP. By default, a request below 1460 bytes, the payload size of an Ethernet maximum transfer unit (MTU) datagram, is sent over UDP. Virtually all HTTP requests fall into this category [23], [8]. 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, the payload of 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. If the server receives a request over a TCP connection and chooses to respond by TCP, the server sends its response over the client-initiated TCP connection.[2]

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 can 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][3].

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

[2] Note that our current prototype does not implement the portion of the protocol that concerns sending requests over TCP.

[3] 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-C.

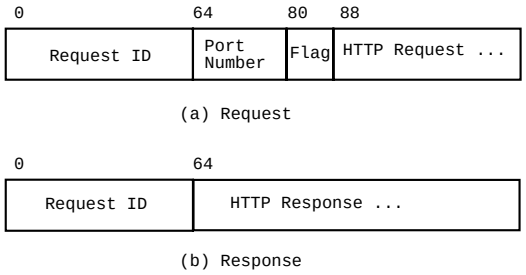


Fig. 2. DHTTP message formats.

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 network traffic for acknowledgments 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 [26]. [4].

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 congestion control (see Section III-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. Since non-idempotent resources are by definition not cacheable, this will not reduce the friendliness of DHTTP to interception caches. Building native support for non-idempotent requests is also possible but would be premature at this stage.

[4]While nothing would prevent the client to disregard this requirement and resend requests in quick succession, this problem is exactly the same with today's Web clients. Fighting these so called *denial of service* attacks is outside the scope of this paper.

There are two ways to delegate non-idempotent requests to TCP. In one method, the protocol portion of a URL would prescribe the transport protocol to be used by clients. (This is the approach used previously by the RTSP protocol [34].) 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.

Alternatively, the protocol can define a special server response that instructs the client to resend its request over TCP. This method does not require the Web site to use special URLs for non-idempotent resources but adds a round-trip delay to the response time when these resources are accessed. As discussed in Section VI-B.3, this response is also useful to improve server resilience to denial of service attacks.

B. Congestion Control

DHTTP servers must avoid flooding a congested network with UDP messages. Instead of implementing its own congestion 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 congestion control for this interaction. An HTTP server using MTU discovery [27] sends packets with payload of 1460 bytes over the Internet and has the initial TCP window for data transfer equal to two packets [5]. Thus, DHTTP server could in principle send up to 2920 bytes by UDP without relaxing TCP's congestion control. Our current default threshold of 1460 bytes makes DHTTP servers even more conservative than HTTP servers in terms of congestion 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. Thus, native TCP congestion control was in place for 94% of interactions. Finally, Feldmann et al. showed that although most Web transfers are short, a majority of the bytes and packets belong to long transfers [16], and DHTTP uses TCP with its native congestion 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.

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

However, when the network is congested, this strategy is extremely poor. Not only do the UDP responses have to be 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.

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.

In summary, the algorithm for choosing a channel is as follows:

1. Choose TCP for all large responses, i.e., whose size exceeds S, as well as for all resent requests.
2. If the ratio of resent request counter to fresh request counter exceeds L, enter a "high loss" mode, else enter a "low loss" mode.
3. In the low loss mode, choose UDP for all small responses, i.e., those below the size threshold S.
4. In the high loss mode, choose TCP for the $1 - L$ fraction of small responses and UDP for the remaining L small responses.

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 grained algorithms, which would track network conditions at at the client autonomous system or even subnet granularity, is an interesting direction for future 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.

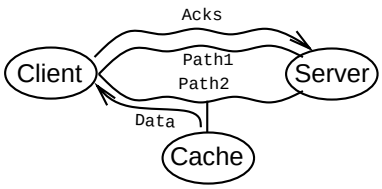


Fig. 3. Problematic deployment of interception cache.

Choosing the 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 (1500 raw bytes or 1460 payload bytes), some responses in the current version of DHTTP will be fragmented. Fragmentation degrades router performance [20]; also, the loss of any fragment will entail resending the entire packet, increasing response latency and bandwidth consumption. Thus, in a high loss environment such as Internet, S should be limited to one MTU. In a low loss network such as a LAN or intranet a higher value of S may be appropriate. Moreover, DHTTP could be extended to allow large responses to be sent over multiple UDP packets. This would avoid fragmentation and allow clients to issue *range requests* [17] to obtain just the missing portions of the response in the aftermath of a packet loss. Exploring this extension and the dynamic policies for choosing the size threshold is another promising direction for future research.

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. 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 acknowledgments 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 acknowledgments 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 principle 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-C 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. Addressing Limitations of HTTP over TCP

Busy HTTP servers face a dilemma of *either* a large number of simultaneous open connections (when using a long persistent connection timeout) *or* a large number of of client requests that pay TCP set-up costs (when using short timeouts). By performing short downloads over UDP, DHTTP reduces the number of TCP connection set-ups *and* at the same time reduces the number of open TCP connections at the server. Furthermore, the TCP connections that DHTTP does open transfer larger objects, increasing connection utilization.

DHTTP also eliminates the head-of-line delay problem of pipelining in existing HTTP. Since it provides the means for explicitly matching of requests and responses, DHTTP has no need for imposing any ordering constraints on server responses. In DHTTP, responses may use UDP or TCP, they may reuse the same TCP connection or use new connections; in any case, the order in which responses are sent is immaterial.

DHTTP effectively addresses another potential problem related to pipelining, first described by Frystyk Nielsen et al. [28]. Unilateral closing of a persistent connection by the server may occur when there are in-transit pipelined requests that the client has sent before receiving the FIN message indicating the termination of the connection. Pipelined requests that arrive to the server after it closed the connection will cause the "reset" message unless the server keeps the receiving half of the connection open. Upon receiving the reset, the client's TCP stack will discard all data that it has received but has not yet acknowledged back to the server. The connection will have to be re-established, and the discarded data requested and sent again.

DHTTP allows either the client or the server to unilaterally close the TCP connection and at the same ensure that the connection has no in-transit data. A client can close a connection to a server when there are no outstanding requests to this server. A server can close a connection to a client at will after sending the current response, since new requests from the client would arrive on the UDP channel or a different TCP connection. While an HTTP server can also address this problem by keeping the receiving part of the connection open [28], DHTTP allows a cleaner solution since it avoids in-transit requests altogether.

C. 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 [11], 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 [24]. 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 [24] and implemented in Firewall-1, which is to prohibit clients from using 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, a successful 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[6]. Second, even if the imposter guessed the port number in time and passed through the firewall, its packets would be

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

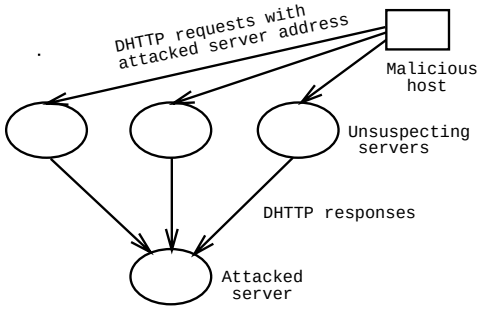


Fig. 4. An induced TCP SYN attack.

discarded by the client because they will not match a valid request ID number (the likelihood of guessing an 8-byte random request ID can be safely dismissed [25]). Learning the correct request ID would require intercepting the request. In fact, it is not inconceivable for a firewall to remove any possibility of letting these requests through by making holes specific to the request IDs. Recent proxy logs showed that over 20 thousand well-connected users generated peak request rate of less than 300 requests per second [42]. Assuming on average it takes a second for the HTTP response to arrive, this would require a firewall serving a 20,000 people enterprise network to maintain a state about only a few hundred requests at a time.

The attacker can also attempt a SYN attack against a DHTTP server using third-party DHTTP servers. In this attack, illustrated on Figure 4, the attacker sends DHTTP requests for large objects to arbitrary other DHTTP servers with the attacked server's IP address as the source address. The contacted DHTTP servers will attempt to respond by opening TCP connections to the specified address and as a result, the attacked server will face a large number of TCP SYN packets from these servers, which unknowingly participate in the attack. Fortunately, the defense used against the active FTP attack applies here as well. Specifically, one must allocate non-overlapping port number ranges to DHTTP servers and clients. Then, if the attacker specifies a low source port number in its requests, the contacted servers will recognize these requests as fraudulent and not act on them. If the attacker specifies a high port number, the SYN packets reaching the attacked server will be addressed to an unexpected port and will be discarded by the server or its firewall.

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. Similar to non-idempotent requests, we assume that clients will choose the TCP channel to send requests for secure downloads. Because the cost of setting up a secure session dwarfs the cost of TCP handshake and because interception proxies do not handle encrypted traffic, there appear to be no tangible benefits in using UDP for either requests or responses, and the current protocol for encrypted Web communication, SSL, does not support UDP exchanges.

D. Network Address Translation

As is the case with firewalls, DHTTP's traversal of NAT (network address translation) devices imposes requirements on these boxes that are similar to many other protocols. A NAT box hides IP addresses of hosts behind it by translating their addresses and port numbers to distinct external addresses and port numbers. To allow unambiguous translation of incoming packets back from external to internal addresses, the NAT box ensures that no two hosts are mapped to the same external address/port pair at the same time.

To support DHTTP, the NAT box must also translate the ephemeral TCP port in outgoing DHTTP requests, and translate back incoming TCP packets accordingly. In other words, the NAT box must be able to translate a new packet flow based on the DHTTP request packet. The ability to consider packet payload in address translation is called *application awareness*, and the ability to translate additional packet flows based on a previous packet is called *stateful translation*. Both capabilities are required for existing protocols and supported to varying extend by most modern NAT devices. An indicative example is voice over IP [22], where a session starts using the Q.931 setup protocol, which specifies ephemeral ports for subsequent flows over TCP (for the call setup) and UDP (for RTP packets that carry the audio itself). Simpler examples include RTSP and active FTP protocols already mentioned in Section IV-C.

In summary, to support DHTTP, a NAT box must be configured to create a temporary mapping from the TCP port number in the payload of a DHTTP request to the internal address of the client that sent the request. For NAT boxes that are not capable of such configuration, an application-level gateway [36] can be configured on the NAT to implement this functionality.

E. 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.

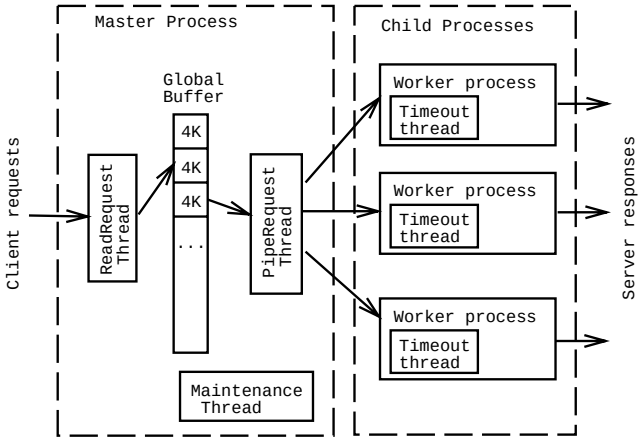


Fig. 5. DHTTP server architecture.

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. Server Design

We have implemented a DHTTP server by modifying Apache, today's most popular Web server. We used Apache 1.3.6 [3] as the basis for our modifications. The structure of our DHTTP server inherits many aspects of Apache and is shown in Figure 5. The server has a master process that accepts incoming requests and worker processes, which execute individual requests and send the responses back to clients. The master process has three threads. A ReadRequestThread thread reads incoming requests from the UDP port and copies them into a global buffer. A PipeRequestThread thread takes the requests from the global buffer and pipes them to worker processes for execution. The global buffer is used to move requests out of the UDP port buffer as quickly as possible, so that the incoming requests will not fill up the UDP port buffer and get dropped. A better alternative, which we have not yet implemented, would have been to increase the size of the UDP port buffer and eliminate the extra copying of requests between the two buffers. Finally, the Maintenance thread wakes up every second and checks the status of the worker processes. If it finds too few idle worker processes, it forks some new ones. If there are too many idle worker processes, it kills some, so that number of worker processes would scale to the request rates.

A worker process reads an HTTP request from its pipe, generates the HTTP response, chooses between UDP and TCP channels and sends the response to the client. If the worker process chose TCP and it already has an open TCP connection to the client, it reuses this connection; otherwise it opens a new connection to the client. Thus, a worker process can have at most one TCP connection to a client, although it may have several concurrent connections to different clients.

Each worker process also contains a Timeout thread that is invoked periodically to close any TCP connections created by this process that have been idle for more than a timeout period. This timeout parameter is equivalent to persistent connection timeout in HTTP servers.

Worker processes share with the master a data structure that describes open TCP connections. The data structure includes a *pending request counter* for each connection, which is an upper bound on the number of requests than may be responded to over this connection. When the PipeRequest thread chooses a worker process and the latter has an open connection to the client, the PipeRequest thread increments the pending request counter (even though it does not know if the response will use the connection or be sent over UDP). When a worker completes sending the response, it decrements the pending request counter for this client.

When choosing a worker process for a new request, the PipeRequest thread tries to efficiently reuse available TCP connections and, at the same time, distribute the incoming requests uniformly among all worker processes according to the following procedure:

1. If there is an open idle TCP connection available to this client and to the TCP port specified in the request message, choose the worker process that has created this TCP connection.
2. Otherwise, if the number of open TCP connections to this client and the total number of open connections are below their respective limits, choose a worker process with the smallest total number of pending requests.
3. Otherwise, choose a process that has a TCP connection to this client with the fewest pending requests; if there are no connections to this client, choose a process with the smallest total number of pending requests.

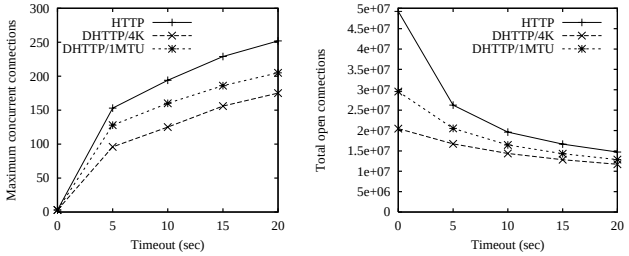
VI. Performance Analysis

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. 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



(a) Max number of concurrent connections.

(b) Total number of TCP connections.

Fig. 6. Number of TCP connections at a server with three connections per client.

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 the server at peak demand [7]. 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 6 and 7 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 [7], [13]. But that increases dramatically the total number of connections (Figure 6b), 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

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

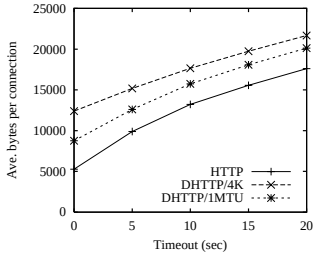
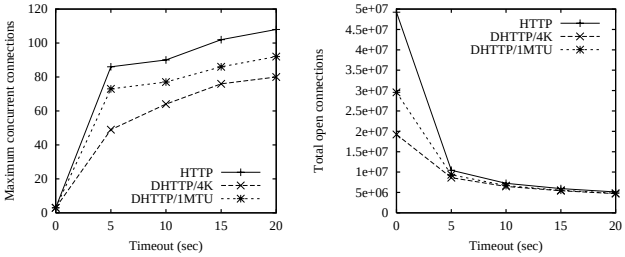


Fig. 7. Connection utilization with three connections per client.



(a) Max number of concurrent connections.

(b) Total number of TCP connections.

Fig. 8. Number of TCP connections at a server with one connection per client.

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 7, 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 8 and 9 depict the results for the case where all clients only use one connection. Figure 8a 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 8b 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 [8]. Figure 9 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.

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 [6]. We found it to be the most appropriate workload generator for our purpose because it mod-

[8] 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.

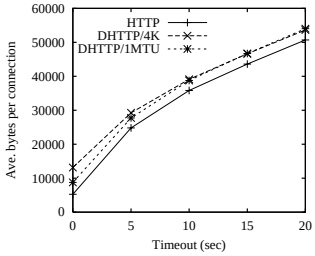


Fig. 9. Connection utilization with one connection per client.

els 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.

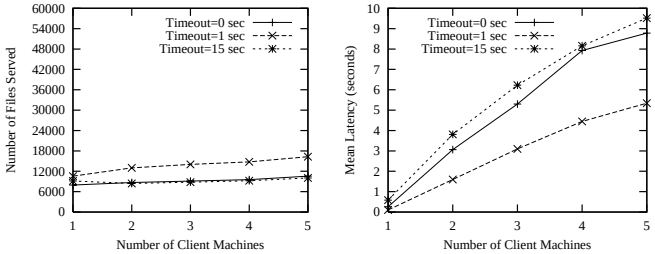
We chose Apache as the basis for our implementation because it is by far the most popular Web server. Recently, more efficient event-based servers have been built (e.g., [39], [43]). We believe our conclusions are not affected by the process-based implementation of Apache because our DHTTP server is 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 (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.

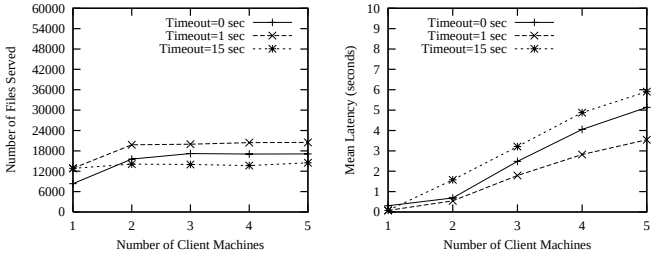
B.1 Bottleneck at Server

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 with 128MB of memory running Linux RedHat 7.3 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.



(a) Throughput with three connections per client.

(b) Latency with three connections per client.



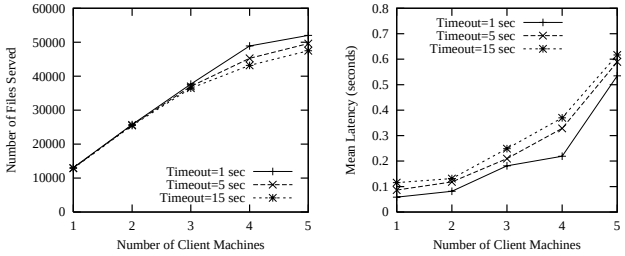
(c) Throughput with one connection per client.

(d) Latency with one connection per client.

Fig. 10. Apache performance (bottleneck at the server).

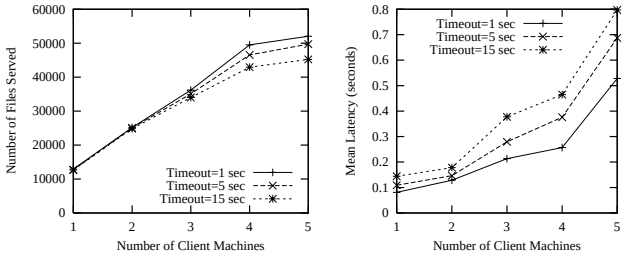
Figure 10 shows the throughput (reported as the number of requests served during the two-minute experiment) and latency of Apache, when the maximum number of parallel TCP connections to a client is set to three and one. The latency in this and all subsequent experiments is client-perceived latency as measured on the client. To compensate for latency hiding due to pipelining, SURGE charges the overlapped latency to only one of the dowdloads. Further, our DHTTP-enabled SURGE issues UDP requests for the page and all its embedded objects at once, thus underestimating the latency to some extent. Because underestimation affects only the initial page download and is amortized over all objects of the page, it should not affect noticeably our results. The figures show curves for persistent connection timeouts of 0 (no persistent connections – the choice of many high-volume sites), 15 (the default value), and 1 second (which we found to produce the best performance). Figure 11 shows the same characteristics for the DHTTP server, for timeout values of 1, 5, and 15 seconds (we did not consider the timeout of 0 because it can lead to a large number of new TCP connections).

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



(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. 11. DHTTP server performance (bottleneck at the server).

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 [7].

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. In the meantime, the number of backlog TCP connections can reach the limit and the server will start dropping TCP SYN requests, sending clients into exponential backoff. 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 as follows. When a worker process is in the blocked *read* call, it checks the TCP *accept queue* every second. Whenever the accept queue is not empty it closes its current connection so that a pending TCP connection 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 and request history indirectly address the same issue [13].

Further, SURGE clients close connections after downloading a certain number of HTML pages, according to the generated schedule. To test a more altruistic client behavior, we modified SURGE to follow a recently proposed *early close* policy [7] where clients close their connections after getting all the files in one HTML page, thereby reducing the number of connections

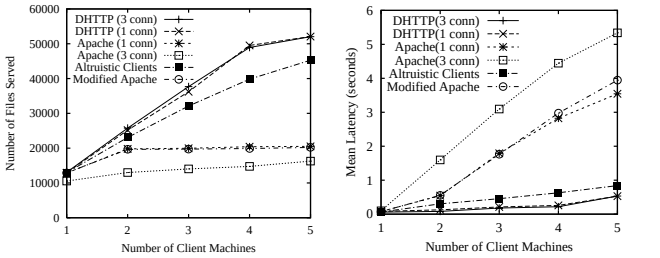


Fig. 12. Comparison of Apache and DHTTP servers: throughput (left) and latency (right).

that servers must handle.

Figure 12 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 10 and 11, 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[9]. 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.

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 13 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

[9] From Figure 12, 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.

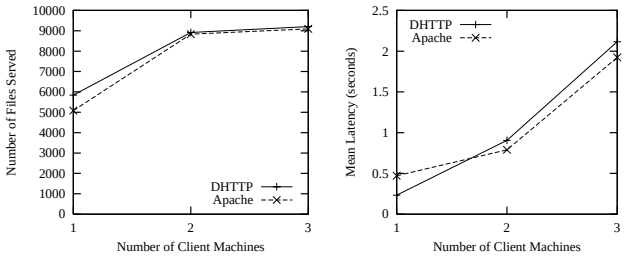


Fig. 13. Apache and DHTTP server performance under network congestion: throughput (left) and latency (right).

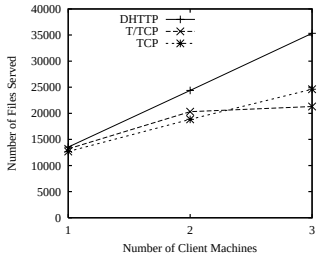


Fig. 14. The throughput of the DHTTP server using TCP and the Apache server using T/TCP.

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%

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.

B.3 DHTTP versus T/TCP

Probably the most closely related existing approach to DHTTP is Transactional TCP (T/TCP) [9], [38], which 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. In the best scenario, the entire HTTP download occurs in three segments: the client's SYN segment that also contains the HTTP request and and the FIN flag (starting the connection, sending data, and closing the connection in one segment), the server SYN/ACK segment that also contains the HTTP response and the FIN flag (at once accepting the connection, sending response and closing the connection), and the client's ACK segment acknowledging the response. Connections from clients that do not have cached state are established using the normal TCP handshake. The T/TCP specification also mentions reusing the congestion window for consecutive connections from the same client similar to Fast Start.

A major functional difference between DHTTP and T/TCP is that, like other transport-layer optimizations, T/TCP does not address the issue of the violation of the end-to-end principle by interception caches and the severe deployment restrictions it entails. Beyond this difference, it is interesting to see how DHTTP and T/TCP compare from purely performance perspective.

Because T/TCP is supposed to be natively supported on FreeBSD (although see [14] for our bug report), we ported our DHTTP server and SURGE to FreeBSD. We further ported the original Apache server (which already runs on FreeBSD) to use T/TCP. Porting Apache required only a minimal change – setting the NOPUSH option on the socket that listens for the incoming TCP connections and closing the connections immediately after servicing all HTTP requests from the incoming segment with the FIN flag set (because the FIN flag indicates that the client will not use this connection for more requests, the server has no no reason to keep this connection open). We then compared the throughput of DHTTP (using standard TCP) and Apache (using T/TCP). Our setup includes four machines running FreeBSD 4.7 connected by a 100Mbps switched LAN: a 731MHz Pentium 3 machine with 2G memory as a server and three 2.8GHz Pentuim 4 machines with 4GB memory each as clients. As before, each client machine runs four processes with 50 threads each, or 200 user equivalents.

To test T/TCP under the most favorable conditions, we used altruistic clients with early close policy to test existing Apache, which showed the highest performance in our previous tests. Furthermore, to factor out a policy decision on whether or not to send the FIN flag with requests for container HTML objects (which would entail, resectively, downloading embedded objects over an extra follow-up connection or incurring an extra overhead for closing the connection if there are no embedded objects), we let the clients send all requests for a page (that is, the container object and all embedded objects) at once, along with the SYN and FIN flags.[10]

Figure 14 plots the throughput of DHTTP and Apache using T/TCP under these conditions. As the basis of comparison, it also plots the throughput of Apache using standard TCP (with altruistic clients, since they showed the best performance for TCP in out tests in Section VI-B.1). We can see that DHTTP achieves much better throughput than Apache-T/TCP. This is due to the fact that T/TCP must still perform all TCP process-

[10] We also tested clients that download the container object and subsequent embedded objects in two separate T/TCP transactions. As expected, these realistic clients achieved slightly, up to 5%, worse throughput than the idealized clients presented in Figure 14.

ing, and because of the bottleneck process in Apache that must handle all the incoming connections. In fact, our tests did not show any appreciable advantage of T/TCP over TCP in terms of throughput. Surprisingly, TCP had actually better throughput at the highest load. We can speculate that this might be due to more careful fine-tuning of TCP performance.

Furthermore, servers are reluctant to support T/TCP for security reasons, and typically either drop the data from a SYN segment[11] or block the entire segment at the firewall. Indeed, to ensure strong TCP guarantees, the server keeps malicious SYN packets for a long time while trying to complete the connection setup: 75 seconds on FreeBSD 4.3 and 45 seconds on FreeBSD 4.6. Allowing the size of this packet to increase from 40 bytes (SYN packet without data) to 1500 bytes (the maximum datagram with MTU discovery) drains server resources that much faster.

Although DHTTP also has to spend resources on processing data from the first packet, the application specifics makes it easier to deal with thet hreat. If the DHTTP server sends a UDP response back, it only deals with this entire request for the duration of processing, typically milliseconds, and maintains no state about this request afterwards. If the server must use TCP and is in a danger zone in terms of overload, it can always respond to the client with a special UDP response asking the client to re-sent the request over traditional TCP and forget about the current request immediately. In this case, the DHTTP server will not open the TCP connection back to the client and forgo the associated advantages, but the only expense compared to existing Web servers is the extra round-trip for legitimate clients, a reasonable price at the time when server survival is in jeopardy.

VII. Future Work

There are a number of ways in which DHTTP could potentially be improved. We discussed throughout the paper a possibility of native support for non-idempotent requests, fine-grained channel selection based on conditions of network connections to individual client autonomous systems or subnets, dynamic policies for size threshold selection, and sending a UDP response over several separate UDP packets.

In addition, 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. Note that DHTTP enables these flexible policies because of server-initiated connections.

VIII. 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.

[11] With an explicit comment in the sources of Linux and FreeBSD 4.7 kernels citing security concerns for discarding the data.

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

From the performance perspective, 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.

The protocol achieves these advantages at the expense of extra requirements on firewalls and NAT devices. Although these requirements are similar to those imposed by an increasing number of other protocols and can be satisfied with an appropriate application-level gateway, they certainly complicate the configuration of these devices. Another limitation of DHTTP is that it leaves out of scope the encrypted Web traffic.

A number of further optimizations to the protocol are clearly possible. However, even the initial minimalist implementation described in this paper demonstrated significant benefits of DHTTP over existing transport. The source code for our DHTTP server implementation, as well as SURGE workload generator modified to use DHTTP, is available [14]. 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, Oliver Spatscheck, Graig Wills, and anonymous referees for useful discussions, comments, and suggestions.

References

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

[2] K. C. Almeroth, M. H. Ammar, and Z. Fei. Scalable delivery of web pages using cyclic best effort multicast. In *INFOCOM*, pages 1214–1221, 1998.

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

[4] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a Web proxy in a cable modem environment. Technical Report HPL-1999-48, HP Labs, April 1999.

[5] P. Barford, A. Bestavros, A. Bradley, and M. Crovella. Changes in Web client access patterns:characteristics and caching implications. *Wolrd Wide Web*, 2(1-2):15–28, Special issue on Characterization and Performance Evaluation 1999.

[6] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of SIGMETRICS-98 Conference*, pages 151–160, 1998.

[7] Paul Barford and Mark Crovella. A performance evaluation of hyper text transfer protocols. In *Proceedings of SIGMETRICS-99 Conference*, pages 188–197, 1999.

[8] Leeann Bent, Michael Rabinovich, Geoff Voelker, and Zhen Xiao. Characterization of a large Web site population with implications for content delivery. In *Proceedings of the 8th International WWW Conference*, May 2004.

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

[10] Barry Brown. U-HTTP: a high-performance UDP-based HTTP. M.S. Thesis. Department of Computer Science and Engineering, UCSD, 1997.

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

[12] Israel Cidon, Raphael Rom, Amit Gupta, and Christoph Schuba. Hybrid TCP-UDP transport for Web traffic. In *IEEE Int'l Performance, Computing and Communications Conference*, pages 177–184, 1999.

[13] Edith Cohen, Haim Kaplan, and Jeffrey D. Oldham. Managing TCP connections under persistent HTTP. In *Proceedings of the 8th International WWW Conference*, pages 631–646, Toronto, Canada, May 1999.

[14] DHTTP. http://www.research.att.com/∼misha/dhttp/abstract.html.

[15] A. Feldmann, R. Cáceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *INFOCOM-99*, pages 107–116, 1999.

[16] Anja Feldmann, Jennifer Rexford, and Ramón Cáceres. Efficient policies for carrying Web traffic over flow-switched networks. *IEEE/ACM Transactions on Networking*, 6(6):673–685, December 1998.

[17] 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.

[18] John Heidemann, Katia Obraczka, and Joe Touch. Modeling the performance of HTTP over several transport protocols. *IEEE – ACM Transactions on Networking*, 5(5):616–630, October 1997.

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

[20] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. In *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology*, pages 390–401, August 1988.

[21] Balachander Krishnamurthy and Martin Arlitt. PRO-COW: Protocol compliance on the web: A longitudinal study. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS-01)*, pages 109–122. The USENIX Association, 2001.

[22] Vineet Kumar, Markku Korpi, and Senthil Sengodan. *IP Telephony with H.323: Architectures for Unified Networks and Integrated Services*. John Wiley & Sons, 2001.

[23] Bruce Mah. An empirical model of HTTP network traffic. In *Proceedings of the INFOCOM'97*, pages 592–600, 1997.

[24] 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*, pages 16–26, 1997.

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

[26] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

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

[28] Henrik Frystyk Nielsen, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Hakon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of ACM SIGCOMM'97 Conference*, pages 155–166, September 1997.

[29] V. N. Padmanabhan and R. H. Katz. Tcp fast start: A technique for speeding up web transfers. In *IEEE Globecom '98 Internet Mini-Conference*, pages 41–46. http://www.cs.columbia.edu/~hgs/InternetTC/GlobalInternet98/, 1998.

[30] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving http latency. *Computer Networks and ISDN Systems*, 28(1–2):25–35, December 1995.

[31] M. Rabinovich and O. Spatscheck. *Web Caching and Replication*. Addison-Wesley, 2001.

[32] P. Rodriguez, S. Sibal, and O. Spatscheck. Tpot: Translucent proxying of tcp. *COMPCOM*, 24:249–255, February 2001.

[33] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

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

[35] M. Spreitzer and W. Janssen. HTTP "Next Generation". In *Proceedings of the 9th Int. World Wide Web Conference*, pages 593–609, May 2000.

[36] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations. RFC 2663, August 1999.

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

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

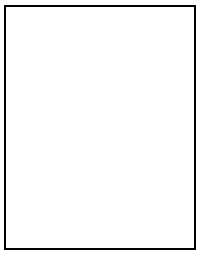
[39] THTTPD - tiny/turbo/throttling HTTP server. `http://www.acme.com/software/ththpd/`.

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

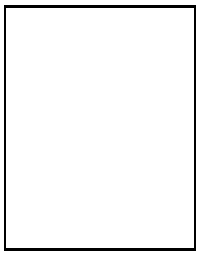
[41] J. Touch. The LSAM proxy cache: a multicast distributed virtual cache. *Computer Networks And ISDN Systems*, 30(22-23):2245–2252, November 1998.

[42] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Molly Brown, Tashana Landray, Denise Pinnel, Anna Karlin, and Henry Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 1999 Usenix Symposium on Internet Technologies and Systems (USITS'99)*, October 1999.

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



Michael Rabinovich Michael Rabinovich (ACM '98) received his PhD in Computer Science from the University of Washington. He is a researcher at AT&T Labs – Research, where he works in the areas of Web and Internet performance, distributed databases, and workflow management. He co-wrote a book "Web Caching and Replication" that appeared in 2002. His email address is: misha@research.att.com.



Hua Wang Hua Wang received his B.S. in mathematics from Xi'an Jiaotong University in China, and his M.S. from Computer Science Department of New York University. He is currently working as an Associate Director for Bear Stearns, Co. Inc. His email address is: wanghua@cs.nyu.edu.