1996

The object delivery transport protocol (ODTP)

Erik Andrew Moore
Lehigh University

Moore, Erik Andrew

The Object Delivery Transport Protocol (ODTP)

June 2, 1996

The Object Delivery Transport Protocol (ODTP)

by

Erik Andrew Moore

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in the Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

May 6, 1996

This thesis is accepted and approved in partial fulfillment of the requirements for the Masters of Science.

May 7 1996

Date

Thesis Advisor

Chairperson of Department

Table of Contents

List of Figures

Abstract

With the increasing use of the World-Wide Web for delivering distributed multimedia documents, the inadequacies of the existing methods of transfer have become readily apparent. This research analyzes the deficiencies of the current HTTP/TCP interaction and demonstrates how existing solutions do not fully address the performance problems of the World-Wide Web. The analysis reveals that existing research into HTTP protocol enhancements and infrastructure research (caching, replication, and multicasting) provide only partial remedies to the WWW's problems. To summarize, HTTP protocol enhancements, including persistent connections and pipelining, reduce the impact of TCP's slow-start congestion avoidance algorithm but do not improve the connection management overhead imposed by TCP. The WWW infrastructure research reduces user latency by reducing the distance an object must travel through the network with caching and/or replication or by increasing scalability through multicasting. These improvements, however, do not affect user latency for a random object regardless of prior and concurrent readership and reveal that a stronger foundation is necessary to provide clients with low latency for a random object. This research establishes three objectives for an efficient transport protocol for the WWW:

1. A transaction rather than stream-based protocol—this implies a low connection management overhead on connection establishment and a minimum transaction latency of one RTT for random objects regardless of prior readership
2. Effective congestion avoidance for high-speed transmission rates and short connection that will allow the protocol to scale with improvements in network bandwidth.
3. Minimization of server load and server processing time (SPT)

The final contribution of this research is the specification of a new transport protocol that addresses these objectives. This new protocol called the Object Delivery Transport Protocol (ODTP) provides efficient connection-oriented, transport services for message transactions between a client and server. ODTP utilizes the naming (port and address) from TCP, timer-based connection management from Delta-t and Fletcher, rate-based flow control from VMTP, and an unique packet response cache that requires a modified HTTP server. A proof of correctness of the protocol's connection management and a formal description of the protocol and the HTTP modifications are provided.

Introduction

With the increasing use of the World-Wide Web for delivering distributed multimedia documents, the inadequacies of the existing methods of transfer have become readily apparent. The HyperText Transfer Protocol (HTTP) is an OSI application level protocol for retrieving the distributed documents in the World-Wide Web. HTTP is generally layered over the reliable connection-oriented Transmission Control Protocol (TCP) transport protocol. HTTP and TCP, though, are an inefficient match because the connection-oriented, stream-based communication of TCP limits the performance of the inherently transaction-based HTTP.

This research analyzes the deficiencies of the current HTTP/TCP interaction and demonstrates that existing HTTP and TCP protocol enhancements, object caching and replication, and multicasting provide only partial remedies to the WWW's problems. Hence, a stronger foundation is necessary to provide clients with low latency for random access regardless of what their prior readership may have been. This research establishes three objectives for an efficient transport protocol for the WW and then specifies a transport protocol underlying HTTP that addresses these objects. The new protocol called the Object Delivery Transport Protocol (ODTP) provides efficient connection-oriented transport services for request/response interactions between a client's browser and an HTTP server.

WWW Architecture

The World-Wide Web (WWW or W3) project was begun by the European Center for High Energy Physics (CERN) in Geneva, Switzerland in 1989 for delivering documents to the High Energy Physics community. Since its inception, the World-Wide Web has become an international distributed hypermedia system. The Web consists of network hosts (or nodes) containing objects (documents) with links to other objects on the same host or on any other host in the WWW. A document is typically a HyperText Markup Language (HTML) file that includes additional graphic files (which also must be retrieved) and links. Each object has a unique Uniform Resource Identifier (URI) which is usually an Uniform Resource Locator (URL) address. Using the client-server paradigm, client software known as a "browser" requests an object using an URL from a WWW server and then presents the object to the user.

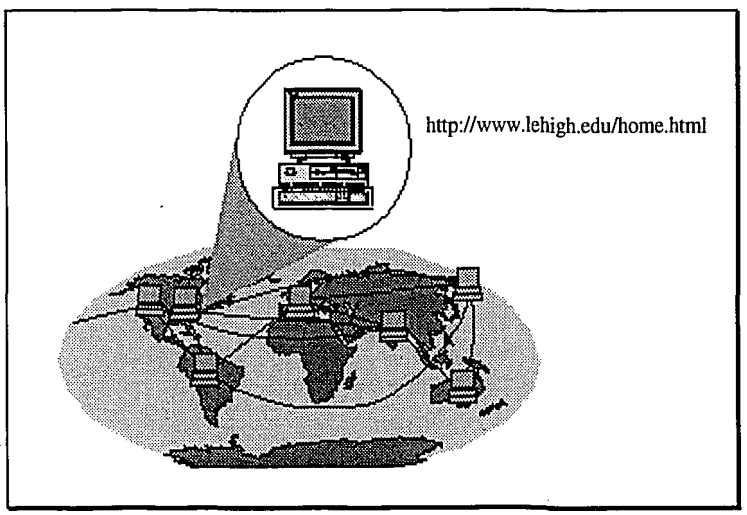

Figure 1: WWW Overview

HyperText Transfer Protocol

The HyperText Transfer Protocol (HTTP) which was developed by Berners-Lee *et al.* (1992, 1994, 1995, and 1996) is used for this communication between the client and server.

HTTP is an application-level protocol (according to the OSI model; see *Figure 2*) that is generic, stateless, and object-oriented.

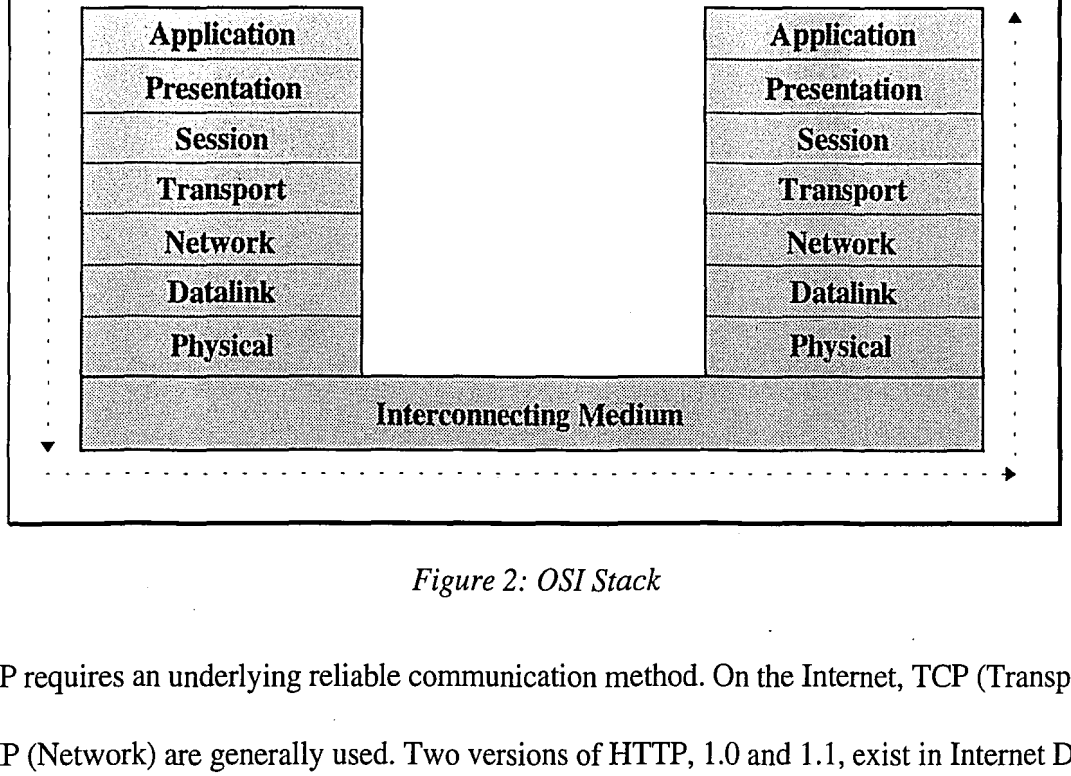

Figure 2: OSI Stack

HTTP requires an underlying reliable communication method. On the Internet, TCP (Transport) and IP (Network) are generally used. Two versions of HTTP, 1.0 and 1.1, exist in Internet Draft form, and both are backwards-compatible with the initial HTTP/0.9 specification (Berners-Lee "HTTP Protocol"). HTTP/1.0 (Berners-Lee "HTTP/1.0") is in widespread use and is used for this discussion.

As described above, HTTP/1.0 uses the request/response paradigm and the following four stages:

1. Connection

   A client first establishes a connection with a server. If TCP/IP is employed as the underlying protocol, a connection is made between the client and the HTTP (TCP) port on the server (usually port 80).

2. Request

The client then sends the server a request with a request method, a Uniform Resource Identifier (typically an URL), a protocol version and a Multipurpose Internet Mail Extensions (MIME)-like message. Allowed request methods are "GET," "HEAD," "POST," and specified extensions. A typical HTTP request also contains a header that includes a number of "Accept" entries that tell the server which object types that the client can handle. These "Accept" entries may detail many more than the sample two illustrated below. An example request for the home page at Lehigh University would be:

```
GET //www.lehigh.edu/home.html HTTP/1.0
ACCEPT: text/plain
ACCEPT: text/html
...
```

3. Response

The server responds with a status line, including the protocol version and a 3-digit success or error code, and a MIME-like response message. The response message from the server will contain the requested HTML document.

```
HTTP/1.0 200
```

4. Disconnect

The server closes the connection after sending the response.

The messages are called MIME-like because HTTP/1.0 reuses many of the constructs defined for Internet mail and MIME but is not a MIME-compliant application.

To illustrate the above, in the simplest request/response form, a client's browser software requests a document on a WWW server and receives the document in the MIME-like message response (see *Figure 3*).

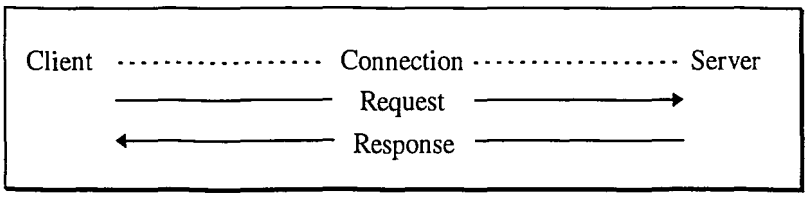

Figure 3: HTTP Request/Response

HTTP also allows intermediaries to be present in the connection path between client and server. An intermediary could be a proxy that forwards messages, a gateway that translates the message, or a tunnel that simply relays the message (see *Figure 4*).

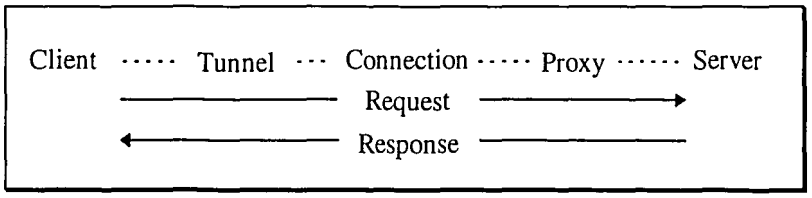

Figure 4: HTTP Request/Response with Intermediaries

Any intermediary except for a tunnel may also cache responses to improve response time. If an intermediary has a cached response for a request, the request will never reach the server (see *Figure 5*).

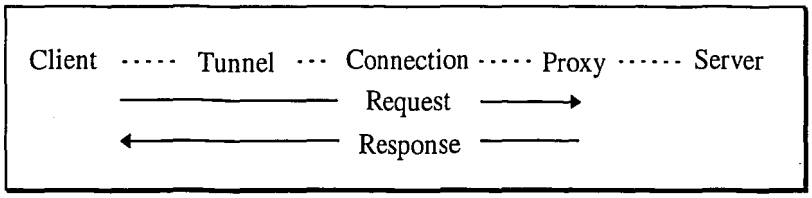

Figure 5: HTTP Request/Response with Proxy Caching

HTTP/1.1 (Berners-Lee, "HTTP/1.1") improves upon HTTP/1.0 by providing persistent connections, hierarchical caching (and identifying when it is unsafe to cache), content negotiation, partial retrieval, request preconditions, digest, and proxy authentication. Despite these improvements, HTTP/1.1's performance is still limited due to its interaction with TCP. Before examining the performance issues of HTTP/1.0, a survey of TCP is useful to understand the interactions between HTTP and TCP.

Transmission Control Protocol

TCP (Postel "TCP") is a connection-oriented (through sockets), reliable, data stream protocol that uses the communication services of a simple (and possible unreliable) datagram

protocol like the Internet Protocol (Postel "IP"). The Internet Protocol (IP) provides TCP with the ability to send and receive variable-length segments through multiple heterogeneous networks and interconnecting gateways. TCP transfers a continuous stream of octets (8-bits) in each direction by packing the octets into segments (the maximum segment size [MSS] can be negotiated—the default is 536 bytes) for transmission through the Internet system. IP then packages and fragments (if necessary) these segments into IP datagrams (which have a maximum size of 65,535 octets although most implementations use 576 octets for remote connections as outlined in Braden ("Requirements")).

To guarantee the order of the transmissions and their reliability, TCP assigns a sequence number to each octet and a checksum to each segment transmitted. TCP then requires the reception of each octet to be acknowledged. The acknowledgments (ACK) may be cumulative so that an acknowledgment of sequence number *X* indicates that all octets up to but not including *X* have been received (Postel "TCP"). If the sender does not receive an ACK for all the octets in a segment within a specified timeout period, it resends the segment. The receiver checks the integrity of the segment by verifying the checksum and uses the sequence numbers to order the segments and eliminate any duplicates. Through these mechanisms, TCP provides a reliable transport of information.

TCP, however, must restrict the flow of information to the receiver because of the receiver's fixed buffer size and processing limitations. A receiver controls the transmission flow by returning with each acknowledgment the number of octets that can be sent before an acknowledgment is received. This "advertised window" (which is limited to 64 kilobytes) indicates the number of unacknowledged octets that the receiver can handle before dropping segments. The sender will stop sending segments when it reaches the limit until an

acknowledgment for a previously unacknowledged segment is received (a receiver in the ACK could also increase the window size up to the maximum size). A receiver can cumulatively acknowledge received segments by returning an ACK of the last consecutive segment received. When a packet is lost, the sender reaches a point where it has sent as much data as the advertised window allows and so it must block and wait for an ACK that will not arrive. Eventually, a timeout (based on the measured round-trip time) will occur. TCP begins retransmissions at this packet that is assumed to have been lost. This strategy, known as *go-back-n*, may lead to the retransmission of segments transmitted to the receiver but not yet received (in the communication channel).

Congestion Avoidance

TCP implementations will retransmit unacknowledged packets several times at increasing time intervals until some upper limit is reached. A sudden load on the network, though, can cause the actual segment round-trip time to rise faster than the sender's measurement of the round-trip time can be updated. As a result, the sender will flood the network with additional copies of the segments. The network will then experience "congestion collapse" as packets are dropped because of full buffers at the routers and gateways and round-trip times reach a maximum (Nagle). To prevent congestion collapse, a sender must not transmit a full window of segments when a network is already congested. TCP implementations handle congestion through slow-start (SS) and congestion avoidance (CA) (Jacobson "Congestion"). These algorithms assume that a packet loss is due to congestion and not damage. For both algorithms, a sender maintains a second "congestion window" (*cwnd*) of the unacknowledged segments. Initially, a sender has

$$cwnd=1. \tag{1}$$

With SS, every time a segment (which may simply be an ACK) is acknowledged without a retransmission,

$$cwnd=cwnd+1; \tag{2}$$

and every time a segment is lost and times out,

$$cwnd=\frac{cwnd}{2}. \tag{3}$$

This procedure results in an exponentially increasing window size. When the first segment is acknowledged, the sender uses (1),

$$cwnd=1+1, \tag{4}$$

and sends two segments. When these segments are acknowledged, the sender increases the *cwnd* to 4 and sends four segments. At some *cwnd* (called the slow-start threshold), TCP implementations switch to using CA. With CA, every time a segment is acknowledged without a retransmission,

$$cwnd=cwnd+\frac{1}{cwnd} \tag{5}$$

and every time a segment is lost and times out,

$$cwnd = \frac{cwnd}{2}. \tag{6}$$

Additional modifications, Fast Retransmit and Fast Recovery, were introduced to change the behavior of TCP when duplicate ACKs are received (Jacobson "Modified").

Three-Way Handshake

To provide this reliability and flow control, the two processes communicating with TCP must establish and maintain a connection and negotiate parameters such as the sequence number and window sizes. TCP uses a three-way handshake with control flags (ACK, synchronize sequence numbers [SYN], and no more data [FIN]) set in the message to establish a connection. A minimal conversation is shown below.

1. The Client sends a Synchronize (SYN) control message with an initial sequence number (ISN) to the Server.
2. The Server sends an ACK of the Client's ISN and a SYN with its own ISN in one control message to the Client.
3. The Client sends an ACK of the Server's ISN to the Server.
4. The connection is terminated by exchanging FIN control messages.

Steps 1-3 establish the connection through the exchange of three messages.

Through the above mechanisms, TCP provides HTTP with a reliable, connection-oriented service for transferring documents in the World-Wide Web. While HTTP/1.0 is designed to be a simple protocol, its inefficiency and interaction with TCP have led to several serious performance problems as the WWW grows.

HTTP/TCP Performance

With an ever increasing number of users, the World-Wide Web faces challenges as access latencies (or response times) continue to rise (Berners-Lee "Propagation," Pam). One problem is "flash crowding" (Pam) in which large number requests from all over the world simultaneously flood a server. A recent example of this problem was the ACM Chess Challenge when World Champion Garry Kasparov played Deep Blue in Chess (February 1996). IBM tried to provide "live" updates of the matches, but its servers were quickly overwhelmed during the first match and additional servers had to be placed online for subsequent matches. Despite these additional servers, the response time for updates was lengthy.

This response time that users experience (or access latency, α) results from both server, client, and network latencies (Padmanabhan and Mogul). Servers (β_s is server latency) and clients (β_c is client latency) with slow hardware (CPUs and disks) will increase access latency. The other component of access latency is network latency that has two factors: bandwidth and propagation delay. Bandwidth (measured in bits per second) is a function of the transmission speed of the network (R). Available bandwidth and the file size (N) contribute to the user response time. Network latency (measured by round-trip time [RTT]) is affected by the distance between client and server (the propagation delay (ρ) with the speed of light a constant) and queuing delays (d) due to network traffic (including cross-traffic).

$$\alpha = \beta_s + \beta_c + \frac{N}{R} + x(\rho + d).$$

(7)

where x is the RTTs used for the transmission. Thus, one strategy to connect the growing number of clients to the WWW servers is to improve latency due to bandwidth ($\frac{N}{R}\downarrow$) and increase server speeds to reduce server latency ($\beta_s\downarrow$). Scaling both the network and servers to meet the demand is possible but prohibitively expensive. As bandwidth and CPU speeds rise, though, the RTT per message exchange $(\rho + d)$ and the number of RTTs (x) necessary for reliable delivery become the dominant factors limiting performance.

To address these performance issues, existing research has focused on relieving network traffic and server loads through caching (per-client and proxy) and the reduction of network latency (reducing the number of RTTs) through improvements to HTTP and TCP.

HTTP/TCP Interaction

Padmanabhan and Mogul and Spero illustrate the interaction between HTTP and TCP that leads to some of these performance issues. As described above, a typical HTTP/1.0 transaction consists of four main components: the establishment of the connection, the request, the response, and the termination of the connection. For the user, the response time is the time between opening the connection and receiving the response. Below is a simple transaction with 2 RTTs in which both the request (REQ) and response (RESP) fit in one TCP segment.

Stage	Client	Server	Comments
1. one RTT	SYN		TCP 3-way handshake
		SYN, ACK	
2. one RTT	ACK, REQ		
		ACK, RESP, FIN	
3. one RTT	ACK, FIN		Not part of user response time
		ACK	

TCP's slow-start will affect the HTTP/1.0 transaction if the request and response data streams are larger than the advertised MSS. TCP must then break the stream into smaller segments. With the initial congestion window of one segment, a request larger than the MSS will require more than one acknowledgment. While the server's congestion window will increase as it successfully acknowledges the receipt of the client's segments, the response may also require more than one acknowledgment. If a server does not advertise a MSS, a default value of 536 bytes is used for remote hosts as detailed in Braden. Below is a transaction with 4 RTTs in which the object request fits into three TCP segments: REQ_1, REQ_2, and REQ_3 and the response fits into three TCP segments: $RESP_1$, $RESP_2$, and $RESP_3$.

Stage	Client	Server	Comments
1. one RTT	SYN		TCP 3-way handshake
		SYN, ACK	
2. three RTTs	ACK, REQ_1		Client's congestion window (*cwnd*) is 1; Server's *cwnd* is initially 1
		ACK	When it receives the ACK, the Client uses slow-start and increases its *cwnd* to 2
	ACK, REQ_2, REQ_3		When it receives the ACK, the Server uses slow-start and increases its *cwnd* to 2
		ACK, $RESP_1$, $RESP_2$	
	ACK		
		ACK, $RESP_3$, FIN	
3. one RTT	ACK, FIN		Not part of user response time
		ACK	

Thus, TCP affects the HTTP/1.0 transaction in the follow ways:

1. When TCP sets up a connection, it sends the connection request to the server and waits for the connection to be accepted or rejected. This procedure adds a delay of one RTT.
2. Slow-start affects the first request on a new connection. If the request will not fit into a single segment (the congestion window's initial size is one

segment), the client must wait extra RTTs before it can finish sending the request. (Slow-start could also affect the first response on a new connection as illustrated above). Jacobson indicates that slow-start is not "that slow: it takes $Rlog_2W$ where R is the round-trip-time and W is the window size in packets" ("Congestion" 2). When dealing with the modest-size files of the WWW, though, the connection might not last enough time for the congestion window to be fully opened.

3. A small MSS aggravates the slow-start algorithm by reducing the segment size RTTs for the segments. Thus, the larger the MSS the better performance until IP fragmentation occurs. With fragmentation, a damaged, delayed, or lost IP packet fragment will cause the entire TCP segment to be undeliverable. Braden ("Requirements") warns that the IP MSS for non-local networks should be 576 bytes to avoid fragmentation in any gateway along the network path.

TCP also affects the server and client load by requiring them to maintain information about the connection (window size and sequence number) for the duration of the connection. Additionally, TCP requires a server to maintain connection information for a specified period of time after termination to be sure the remote TCP has received the acknowledgment of its connection termination request.

TCP Performance Research

Researchers have proposed improvements to several aspects of TCP that will also affect HTTP's performance. These areas include more accurate round-trip time measurements, selective and/or negative acknowledgments, larger window sizes, and proactive congestion detection.

As Nagle and Jacobson ("Congestion Avoidance") discuss, without stable flow of packets into and out of the network at the same rate, networks can succumb to congestion collapse. TCP's current implementation of slow-start, though, as outlined by Jacobson ("Congestion Avoidance") is reactive rather than proactive. The algorithm uses a loss of a

segment as a signal that there is congestion in the network but has no means of preventing congestion through detection and adjustments to the sending rate.

Proactive Congestion Detection

Current approaches for proactive congestion detection include Jain's CARD (Congestion Avoidance using Round-Trip Delay), the Tri-S Scheme (Wang and Crowcroft), and TCP Vegas (Brakmo *et al.*). In general, these procedures adjust the sending rate based on comparisons of RTTs or throughput. Using TCP Vegas's algorithm (Brakmo *et al.* 30),

$$Throughput_{expected} = \frac{cwnd}{RTT_{base}} \text{ and} \tag{8}$$

$$Throughput_{actual} = \frac{N}{RTT_{sample}}, \tag{9}$$

where N is the number bytes transmitted between the time the segment was sent and the ACK was received for the segment, RTT_{sample} is the difference between the time the segment was sent and the ACK was received, and RTT_{base} is the minimum RTT_{sample}.

$$Throughput_{diff} = Throughput_{expected} - Throughput_{actual} \tag{10}$$

where $Throughput_{diff} \geq 0$ and α and β are thresholds. Brakmo outlines two rules for determining the current condition of the throughput:

if $Throughput_{diff} < \alpha$
  then not utilizing bandwidth
end if

if $Throughput_{diff} > \beta$
  then congestion
end if

where α and β are thresholds.

Pink, however, indicates that all these algorithms fail if packets are lost for bit errors. Jacobson argues that "loss due to damage is rare (<<1%)" and slow-start will only take $\frac{w^2}{3}$ packets to regain the window's original size (w) after a packet loss ("Congestion" 4). Mishra *et al.* propose a method of distinguishing random loss from congestion loss. They create a heuristic that regards a loss as random if it is the only loss in the window otherwise it is a regarded as congestion related. As shown above, slow-start has a significant impact on HTTP's performance because of the modest-sized files typically requested. The suggested improvements, though, will not affect slow-start's initial impact on HTTP but will affect its performance if a packet is damaged or lost by not closing the window.

Increased Maximum Window Size

Another aspect of TCP that affects HTTP's performance is the limit on the window size. Jacobson *et al.* ("Extensions for High Performance," "Extensions for Long-Delay Paths") illustrate how the 16-bit field of TCP transmission window limits the effective bandwidth to $\frac{2^{16}}{RTT}$ and proposes increasing the window size to 30-bits using a scale option. TCP reliability, however, depends upon the Maximum Segment Lifetime (MSL) (time is bounded by Time To Live (TTL) field of IP datagrams) to prevent sequence numbers from an earlier incarnation (same hosts and sockets) of a connection from affecting the same connection. Sequence numbers will be reused because the sequence number field is finite at 32-bits. McKenzie explains that while sequence numbers will wrap around after 65,536 (2^{16}) RTTs with a 2^{16} window size, they will wrap around after only 4 (2^2) RTTs with a 2^{30} bit window size. Jacobson *et al.* ("Extensions

for High Performance") adds that high-speed connections alone can cause a sequence number reliability problem. A Gigabit (1 Gbps) connection will wrap sequence numbers in only 17 seconds. Because TCP assumes a MSL of 120 seconds (Postel "TCP"), the reliability of sequence numbers cannot be guaranteed. For HTTP, the limitation of the 16-bit window size will harm performance as network bandwidths increase.

Selective and Negative Acknowledgments

TCP's cumulative acknowledgment mechanism only acknowledges a segment when it has successfully received all the segments up to and including the segment. As a result, a receiver has no means of informing a sender that it has received and buffered a segment that is received out of order. With cumulative acknowledgments, when a segment is lost, any segment following it will not be acknowledged (even if it has been successfully received) until the preceding segment is acknowledged. The sender, consequently, could timeout waiting for the acknowledgments and unnecessarily retransmit the segments. Mathias *et al.* and Jacobson *et al.* ("Extensions for Long-Delay Paths") propose improving the cumulative acknowledgments of TCP by adding selective acknowledgments (SACK) to inform senders of non-contiguous blocks of data that have been received (their implementations are limited to 10 blocks in one TCP segment). Fall and Floyd demonstrate through simulation both the strength and weaknesses of SACKs with TCP. One key issue with SACK is the additional buffers required for storing the out-of-sequence packets.

Fox discusses one of the disadvantages of selected acknowledgment schemes namely that a receiver will wait a period of time before sending the selective acknowledgments so they may be bundled together. This delay will underutilize bandwidth. Selective acknowledgments

also require complex state information to determine which packets have been acknowledged. Fox offers an alternative method of negative acknowledgments (NACK) in which the notification of a missing segment is made one at time. Fox argues that while SACKs are more effective than NACKs when packets are lost close together, empirical studies have demonstrated that most lost packets occur far enough apart that SACK's advantage is negligible. Improvements to the cumulative acknowledgment mechanism of TCP are important to HTTP because it better utilizes the available bandwidth by not unnecessarily retransmitting segments. NACKs are superior to SACKs because of their simplicity and low segment and server overhead.

Round-Trip Time Measurement

Another aspect of TCP that affects the number of packets possibly unnecessarily retransmitted is the measurement of the round-trip time. This measurement is used by TCP's retransmission timer to determine when to assume a packet has been lost and must be retransmitted. The accuracy of this timer will significantly affect TCP's performance (Pink). If the timer is too short, bandwidth is underutilized as packets are retransmitted that have already arrived but whose acknowledgments have not been received. If the timer is too long, bandwidth is also underutilized as the sender waits for acknowledgments that will not arrive. To determine the timer value, TCP must estimate the RTT by using a transmission timer.

Jacobson ("Berkeley TCP"; "Congestion" 17) presents an analysis of the round-trip estimator that differs from the original RTT measurement in the TCP specification (Postel "TCP") by including both mean and variance. First, consider the mean RTT,

$$R = (1-g)R + g(M), \tag{11}$$

where R is the smoothed RTT (an estimator of the average), M is the round trip measurement from the most recently acknowledged data packet, and g is the filter gain that is related to the variance of M. Rearranging the terms,

$$R = R + g(M-R), \tag{12}$$

where $M-R$ is the prediction error. Thus,

$$R = R + gE_r + gE_e, \tag{13}$$

where E_r is the random error to unpredictable effects and E_e is the error due to poor prediction. Jacobson suggests choosing a small g (0.1-0.2) to get mileage out of E_e but also to minimize the damage of E_r. For an approximation of the standard deviation, Jacobson uses the smoothed mean deviation which is easier to compute,

$$D = D + h\,(|M-R| - D). \tag{14}$$

Hence, the retransmit timer (RTO) which incorporates both mean and variance is

$$RTO = R + \beta * D, \tag{15}$$

where RTO is the retransmit timer, and β accounts for RTT deviation. Jacobson originally suggested a β of 2 (Jacobson "Congestion") but later recommended a β of 4 (Jacobson "Berkeley TCP"). Jacobson then develops a fast estimator of RTT ("Congestion" 17) using integer only arithmetic and chooses a g of 0.125 and h of 0.25.

To measure RTT rather than estimate RTT, Jacobson *et al.* ("Extensions for Long-Delay Paths") propose adding an echo option to TCP in which a sender places a timestamp in the segment and the receiver returns that time stamp in the corresponding acknowledgment. The

difference is an accurate RTT. This improvement to TCP is significant because TCP requires an accurate measurement of the average round-trip time. An inaccurate estimate of RTT will degrade the performance of HTTP by waiting too long for acknowledgments that will not arrive and by retransmitted segments that have already been received but have not been acknowledged.

HTTP Performance Research

HTTP/1.0 itself adds additional RTTs by forcing connections to be reestablished between requests and by allowing only one request per transaction—this procedure is problematic because a typical HTML file contains many inline graphics. As a result, in the simplest case of one HTML document and an inline graphic file in which both the requests and responses fit in one TCP segment, each transaction will take 2 RTTs (for a total of 4 RTTs).

HTTP/1.0 Protocol Enhancements

Padmanabhan and Mogul have proposed pipelining and persistent connection improvements to HTTP/1.0 to reduce the number of round trips necessary for multiple transactions. With long-lived or persistent connections, a single connection may be utilized for multiple HTTP transactions. Thus, the connection may stay open for all the inline images of a single document. An added benefit of persistent connections is a reduction in a server's load. Because a server normally forks a new process for each request, multiple requests over a single process will eliminate the costs involved with forking new processes. Additionally, less connection state records will be needed (and held onto in the TIME_WAIT delay—see explanation below). To further reduce the number of RTTs, they suggest a request method called GETALL that causes the server to parse the document and return all the objects included in the original requested object (document). GETALL increases the server's load with the parsing step,

but the parsed information could be cached for future requests. The pipelining of HTTP requests with a persistent connection in which the same two objects (Requests REQ_1 and REQ_2 each fit in one segment; Responses $RESP_1$ and $RESP_2$ each fit in one segment) are requested as above improves the situation to 2 RTTs.

Stage	Client	Server	Comments
1. one RTT	SYN		TCP 3-way handshake
		SYN, ACK	
2. one RTT	ACK, REQ1, REQ2 ("GETALL")		Client's *cwnd* is 1; Server's *cwnd* is 2
		ACK, RESP1, RESP2, FIN	
3. one RTT	ACK, FIN		Not part of user response time
		ACK	

Their experimental results using a modified Mosaic v2.4 client and both a local (10 Mbit/sec Ethernet) and a remote (1.544 Mbit/sec T1 link) NCSA httpd v1.3 server reveal that:

1. With the number of inline images ranging from 0 to 10 with size 2,544 bytes, the new protocol with pipelining had a 50% improvement over the original protocol.
2. With the number of inline images ranging from 0 to 10 with size 45,566 bytes, the new protocol with pipelining had a 22% improvement over the original one.
3. Improvements were more significant for remote servers due to the reduction in round trips rather than reduction in per-connection overheads.

HTTP/1.1

HTTP/1.1 (Berners-Lee, "HTTP/1.1") resolves one of these issues by supporting persistent connections. Pipelined requests, while not supported directly, are also allowable according to the functional specification. These improvements lessen the impact of slow-start by maintaining the connection for a longer period of time but do not address the performance limitations of matching the transaction-style communications of HTTP to the stream-based, connection-oriented TCP.

WWW Infrastructure Research

As described below, researchers have also examined improvements to the infrastructure of the World-Wide Web including multicasting, replication (with a wide area filesystem) and caching (both per-client and proxy).

Caching

Because of HTTP's inherent inefficiency when faced with frequent retrievals of the same object (i.e., a duplicate request), researchers as discussed below have suggested that both server and network loads can be alleviated by caching documents with per-client and/or proxy caches. With a per-client cache (which may be persistent or non-persistent), a copy of an object retrieved by a client's web browser is placed in a local cache. When a client accesses an object, the web browser first checks the local cache. If the object is not found, the browser then accesses the file through the network. With per-client caching, web users experience shorter delays when requesting an object and the network sees less traffic. A proxy server caches copies of popular objects on servers that are closer to the clients. A proxy server generates a cache hit when the same user requests an object two or more times or two different users request the same object. These second-level caches get only the misses left over from the web clients that use a per-client cache. A proxy cache, though, saves disk space by not maintaining a copy of the same object on multiple workstations. Both these caches may maintain consistency by issuing a conditional GET method or using a time-to-live for cached files. A conditional GET transaction, however, suffers the same inefficiencies as discussed above. Various authors (Glassman, Smith, Luotonen and Altis) initially examined proxy caching but the following authors provide expanded analysis.

Pitkow and Recker study the access of objects in the WWW using psychological research on human memory to develop an analytical foundation for proxy caching strategies. Using data obtained over a three month period from the Georgia Tech WWW repository and 7-day window (the seven previous days are used when analyzing the accesses made on the eighth day), they determine that the frequency and recency of past documents accesses are both strong predictors of future access, but recency (92% of the variability in access probability is explained by recency) has more of an impact than frequency (72% of the variability in access probability is explained by frequency). They suggest using the classic Least Recently Used (LRU) for object replacement when the proxy cache is full.

Abrams *et al.* analyze cache performance and replacement strategies using the simulation of three workloads. They determine that LRU is not the best policy because when the cache is full and a document is replaced, LRU does not consider the size of the objects. They contend that if the cache size is limited, the replacement of many small files with one large file is not the best strategy. A better strategy, they call LRU-MIN, applies LRU only to the largest documents and then to groups of successively smaller documents. Abrams *et al.* note that the proxy cache hit rate tends to decline with time because browsers use their own caches and the proxy acts as a second level cache. They determine an upper bound of 30-50% on the proxy cache's hit rate given an infinite cache and an eight day interval. Thus, despite these strategies, 50-70% of all objects accessed through a proxy server will still result in accesses through the network.

In analyzing server traces at the School of Computer Science at Carnegie-Mellon University, Spasojevic *et al.* also conclude that there exists only a small set of cacheable documents that they term the "hot set." They contend that information browsing also exhibits

many read-once patterns. An effective caching strategy should then be proactive rather than reactive and prefetch certain objects based upon usage statistics. Even though the client's cache may be satisfying requests that will never reach the proxy server, caching has an overall limited impact on reducing network traffic. Additionally, caching strategies are not effective for dynamic WWW pages with updated sports scores, news, and query responses. To accommodate the growing number of users on the WWW that caching alone cannot manage, Spasojevic *et al.* propose a wide-area file system for the WWW with object transparency, migration, and replication.

Replication and Migration

As Spasojevic *et al.* indicate, the WWW still suffers from the "flash crowding" problem (discussed above) with proxy and per-client caches. Thus, Spasojevic *et al.* propose a location independent wide-area file system (their system uses AFS). With a wide-area file system, information can be migrated from a busy server to an idle server to balance the load. Objects can also be replicated across several file servers. The challenge of using a wide-area file system is that objects in the WWW are specified by a Universal Resource Locator (URL) which specifies one and only one object. If an object is moved, the URL will continue to point to the old location and not the new one.

Multicasting

Another method for reducing network and server loads is multicasting. With Clark and Ammar's multicasting research, requests for the same object can be grouped together and the response can be transmitted using a single multicast connection between the server and multiple clients. This procedure reduces both server and network loads. A key issue with multicasting is

the time to wait for additional requests for the object to arrive at the server so the response can be multicasted rather than unicasted. Clark and Ammar's results indicate a 50% improvement over unicast connections with 20 users using a local network. With less than 10 simultaneous users, though, they found that unicast connections were slightly faster because of the delays involved with grouping the requests. Multicasting for the WWW is an important research area for improving performance when faced with "flash crowding." With the read-once pattern of many clients, though, multicasting is not a complete solution to the performance issues of the World-Wide Web.

Problem Statement

The above analysis reveals that existing research into HTTP protocol enhancements, caching, replication, and multicasting provide partial remedies to the WWW's problems. To summarize, the HTTP protocol enhancements, including persistent connections and pipelining, lessen the impact of TCP's slow-start congestion avoidance algorithm but do not improve the connection management overhead (in the synchronization of ISNs) imposed by TCP. The WWW infrastructure research improves user latency by reducing the distance an object must travel through the network with caching and/or replication or by increasing scalability through multicasting. These improvements, however, do not affect user latency for a random object regardless of prior and concurrent readership. Thus, a stronger foundation is necessary to provide clients with low latency for a random object. The goals for a new transport protocol to address the problems of HTTP/TCP include:

Objectives

1. A transaction rather than stream-based protocol—this implies a low connection management overhead on connection establishment and a minimum transaction latency of one RTT for random objects regardless of prior readership
2. Effective connection avoidance for high-speed transmission rates and short connection that will allow the protocol to scale with improvements in network bandwidth.
3. Minimization of server load and server processing time (SPT)

Transaction-Based Transport Protocol

To satisfy the first objective above (**OBJ1**), a transport protocol must be chosen that minimizes the connection management overhead while providing a transaction-based transport service.

Transport Protocol Overview

The WWW requires a transport layer protocol similar to TCP that provides a connection-oriented, reliable end-to-end transfer of data over potentially unreliable lower layer protocols. (If the lower layer protocols guarantee reliability, the transport protocol's complexity is significantly reduced). A reliable connection-oriented protocol provides a data stream service and/or a transaction-oriented service for client/server interactions (a request followed by a response). TCP, for example, is a data stream service. With a transaction-oriented service, a transaction begins with the transmission of a request from a client to a server and terminates with the response from the server. A transaction-oriented service typically does not have any explicit connection establishment procedures.

Reliability

Overall, there are three fundamental criteria of reliability for these transport protocols (Doeringer *et al.*):

1. In-sequence delivery
2. Complete delivery
3. Freedom from errors

To provide the in-sequence and complete delivery of user data, transport protocols typically use sequence numbers to detect missing or out-of-sequence data. For bit-error detection, the protocols use a checksum of the entire segment (the Transport Data Protocol Unit [TPDU]) or the header and data separately. Some protocols use error reporting with negative or selective acknowledgments to inform the sender of missing or out-of-sequence data. For high-speed networks, NACKs provide the receiver with the necessary error reporting with little overhead. Transport protocols correct errors by using positive acknowledgments with retransmissions (PAR) [TCP uses PAR] or automatic repeat request (ARQ) in which the receiver informs the sender that the data must be retransmitted. Because ARQ limits the additional traffic introduced into the network, ARQ is superior to PAR.

Flow Control

Transport protocols must also maintain end-to-end flow control to guarantee data is transmitted at a rate that the receiver can receive and process the data (without buffer overflow and data loss). Flow control has an increased importance for high-performance networks in which a link could supply data faster than the receiver can consume it (Doeringer *et al.*). The network itself also imposes a restriction on the rate of transmission. Thus, a sender must

maintain access control to avoid network congestion caused by exceeding the data-rate of the network.

TCP, described earlier, uses a window or credit-based flow control. When the window is full, new data will not be transmitted until a "credit" or acknowledgment is received. As transmission rates increase and the whole window can be transmitted, the sender must remain silent and wait at least 2 propagation delays for new credits to arrive. Thus, as Dabbous indicates, for window-based flow control to be effective, the window size (W) must be

$$W > \frac{2DC}{L}, \tag{16}$$

where D is delay, C is the capacity of the link in bits per second, and L is the size of the packets in bits.

With the small initial window size due to slow-start, the resulting throughput is less than the potential throughput even in the absence of congestion. Thus, window-based flow control in which transmissions start and stop according to the demands of the flow control window is inefficient in networks with high transmission rates and/or long propagation delays.

To provide access control to avoid network congestion, TCP uses estimates of the round-trip delay for detection and then slow-start for avoidance. Zhang *et al.* describe a problem with using ACKs as credits that is known as "ACK-compression." With ACK-compression, acknowledgments coming back to the senders are caught behind data in the gateway queues. Thus, they claim it is invalid to assume that acknowledgments always provide a reliable clocking method for data transmissions. Grouping of these acknowledgments also leads to bursty traffic on the network.

Instead of relying on credit-based flow control in which the sender must block waiting for new credits, rate control maintains the flow of packets to the receiver through the intermediate nodes by modifying the rate of transmission and the size of the data (burst) transmitted. A sender must know the maximum rate at which data can be transmitted (to not overflow the receiver's buffer and/or the networks and gateways) and the maximum burst size. A sender can directly specify the transmission rate and burst rate, or the sender can control the flow of packets by estimating and adjusting the time between packet transmissions (the interpacket gap time). For reduced overhead and better performance, these adjustments may take place between groups of packets. Determining the initial values for the burst size and transmission rate and the adjustments necessary for rate control, though, is difficult without network support. With current and future networks using ISDN and ATM that enforce access control based on data rates, rate control will become an ideal flow-control mechanism (Doeringer *et al.*).

Connection Management Requirements for Protocol Correctness

For connection-oriented communications, the transport layer protocol is responsible for setting up the connection, reliably transferring the data, and then destroying the connection. The protocol must protect against the connection management hazards introduced by lost, duplicated, or out-of-sequence packets. Watson ("Delta-t" 7-8) outlines the connection management requirements for transport protocols:

General (G)

1. An identifier of an information unit used for error control must not be reused while one or more copies of that unit or its ACK are alive.
2. The error control information must itself be error corrected.
3. If the crash of an end can cause it to lose its state, then appropriate crash recovery mechanism must assure the other requirements (**G1**, **G2**, **O1**, **O2**, **C1**, **C2**, and **C3**).

Connection Opening (O)

1. If no connection exists and the receiver is willing to receive, no duplicate packets from a previously closed connection should cause a new connection to be established and duplicate data to be accepted unless the operations represented by the data are known to be idempotent.
2. If a connection exists, then no packets from a previously closed connection should be acceptable within a current connection.

Connection Closing (C)

1. No packet from a previous connection should cause an existing connection to close.
2. A receiving side should not close until it has received all of a sender's possible retransmissions and can respond to them.
3. A sending side should not close until it has received acknowledgment of all that it has sent. In particular, it should allow time for an acknowledgment of its final retransmission, if needed, before reporting a failure to its client program.

TCP and the Connection Management Requirements

For example, a handshaking transport protocol like TCP satisfies **O1** with its three-way handshake in which each side sends an initial sequence number (ISN) and acknowledges the other's ISN for the conversation. Through this procedure, the two TCPs synchronize on each other's ISN. The sequence numbers prevent packets that are delayed in the network from being delivered late and then misinterpreted as part of an existing connection. The ISNs are based on a 32-bit counter whose value is incremented every 4 microseconds and cycle every 4.55 hours (Postel "TCP"). To prevent an ISN from being confused with a SN from an earlier incarnation of the connection (there is a non-zero probability of the ISN chosen being identical to the current

SN that exists in the network), TCP specifies a "quiet time" interval of one MSL upon start-up after a crash to ensure the packets have expired. This "quiet time" is unnecessary if the start-up time for the host exceeds one MSL. These ISNs also provide crash recovery (**G3**).

TCP addresses **G2** by applying the CRC to the entire TDPU.

O2 and **C1** require the sequence numbers from one incarnation (same hosts and sockets) of a connection from not being used while the same sequence numbers still exist in the network from an earlier incarnation. TCP depends upon selection of unique ISNs and the Maximum Segment Lifetime (MSL) to guarantee a packet cannot exist in the network after a specified length of time. This time is bounded by the Time To Live (TTL) field of IP datagrams to prevent a violation of **O2** and **C1**.

To ensure that sequence numbers do not wrap within a connection and thus, satisfy **G1**,

$$\frac{|SN|}{B} > MSL, \tag{17}$$

where $|SN|$ denotes the size of the sequence number space and B is the bandwidth of the connection in bytes per unit of time. Recall, each byte (octet) transmitted requires a sequence number. The 16-bit window of TCP, though, effectively limits B to

$$\frac{2^{16}}{RTT}. \tag{18}$$

Thus, if the RTT is large enough, Jacobson *et al.* state that (17) is satisfied ("Extensions for High Performance"). As bandwidth increases, (17) and consequently, **G1** may fail.

To satisfy requirements **C2** and **C3**, the sender of the last close-ACK must wait an interval (in the TIME_WAIT state of TCP) to guarantee it can resend the final ACK in case it is

lost. TCP remains in the TIME_WAIT state for 2*MSL. Remaining in this state requires TCP to maintain a connection's information for a period of time after the connection is terminated. Researchers have improved TCP's handshake mechanism and reduced the time spent in TIME_WAIT by caching certain information from one incarnation to the next.

Transaction-Oriented Extensions to TCP

Braden ("T/TCP") proposes a TCP extension for an efficient transaction-oriented transport protocol. T/TCP (Transaction TCP) improves upon TCP by bypassing the 3-way handshake (in certain situations) and reducing the delay in the TIME_WAIT state. Braden uses a 32-bit "connection count" (CC) in T/TCP that is incremented monotonically in successive connections to distinguish connections and prevent segments from an earlier incarnation of the same connection from affecting the existing one. T/TCP again relies on the MSL to guarantee that two identical CCs values (one from wrap around) for the same connection cannot exist in the network at the same time. A T/TCP server caches these CC values for use in later requests (if any). This caching eliminates the 3-way handshake while still satisfying **O1**. Shankar *et al.* mathematically determine the minimum bound on the incarnation numbers and the optimal residency time. By comparing the cached CC value with the CC value received in the SYN connection request segment, the T/TCP server can distinguish a new incarnation of a connection (it has a larger CC than the cached CC) from a duplicate or out of order connection request. If the T/TCP implementation determines that the connection request is not a new request, it falls back to the normal 3-way handshake of TCP. Because the server can distinguish incarnations of a connection through these cached CC values, the TIME-WAIT delay can be reduced by

allowing a new incarnation of a connection before the TIME-WAIT has expired. A minimal conversation is shown below:

Stage	Client	Server	Comments
1. one RTT	SYN, REQ, FIN		T/TCP
		SYN, ACK, RESP, FIN	
2. one RTT	ACK		Not part of user response time

Thus, T/TCP partially addresses **OBJ1** but worsens **OBJ3** by requiring cached information for each unique connection (unique host and socket). T/TCP does not fully satisfy **OBJ1** because a random object access could result in a fallback to the 3-way handshake if the cached value is unknown (from a crash or no prior access) or has wrapped around.

Timer-Based Transaction-Oriented Transport Protocols

An alternative to handshaking protocols is a timer-based protocol such as Delta-t (Watson "Delta-t", Fletcher). With a timer-based protocol, the connection is established when the first packet is received. This mechanism eliminates the minimum one round-trip time required by the setup mechanism of a handshaking protocol and thus, reduces the round-trip time for a minimal conversation to only 1. A deficiency in timer-based protocols is that they do not allow the negotiation of parameters during connection setup so default ("safe") values must be chosen (but they be modified later).

Delta-t

Delta-t relies on the network layer for guaranteeing a packet will not exceed its time-to-live—a time-to-live field is decremented during routing, retransmission, and acknowledgment to ensure that the lifetime of a packet is bounded (similar to the MSL of TCP). Watson ("Delta-t" 11) outlines the connection management requirements for a timer-based protocol.

The receiver maintains a receive-timer that is started each time a new connection is made. To meet requirements **O1**, **O2**, **C1**, and **C2**, the receive-timer will expire only after all sender retransmissions and other duplicates have a chance to be recognized. If a receiver crashes and loses state, it must wait a specified period of time (which should be greater than the sender's "give-up" timer plus the maximum TPDU lifetime) before accepting a new connection to meet **G1**, **G3**, **O1**, **O2**, and **C1** in order for retransmissions and duplicates sent before the crash to expire.

The sender maintains a send-timer ("give-up") that is activated upon transmission of a segment. Requirements **G1** and **C3** demand that this timer expire only after all the data that has been transmitted or retransmitted has a chance to be acknowledged. If a sender crashes, it must wait a specified period to satisfy requirements **G1** and **G3**. Thus, a timer-based protocol addresses **OBJ1** (transaction-oriented and RTTs are the same for random access) and **OBJ3** (server just maintains a timer). Delta-t maintains flow control by using a sliding window and provides error reporting through negative acknowledgments with *go-back-n* retransmission. The chief problem with a timer-based protocol such as Delta-t is finding accurate values of the timers using network delays and packet lifetimes (Tawbi *et al.*). The heterogeneous Internet makes determining these values difficult. Inaccurate timer values will either be inefficient because they expire after all the packets from the connection have left the network or violate **G1**, **O1**, **O2**, **C1**, **C2**, and **C3** by expiring before all the data that has been transmitted or retransmitted has a chance to be acknowledged.

TP++

Another example of a timer-based protocol is Feldmeier's TP++, which is a transport protocol for multimedia applications that operates across heterogeneous high-speed networks

("TP++") but relies on the network for congestion control. The connection management protocol used by TP++ is Connection Management with Synchronized Clocks (CMSC) (Biersack and Feldmeier). Using synchronized clocks, a TP++ sender timestamps each segment (TDPU) with an expiration timer. The receiver then discards TPDUs that arrive after their expiration time. The expiration time is maintained on retransmitted TPDUs. Each host maintains an ε-synchronized clock (ε bounds the clock skew) that is kept synchronized by executing a clock synchronization protocol such as the Network Time Protocol (Mill). Each connection is assigned a Connection Identifier (CID) by the sender and every message carries this CID. To satisfy **O2** and **C1**, a CID must not be reused while there exists a connection record with the CID at the receiver or messages exist that carry this CID.

TP++ has two timers at the receiver (T_{R1} and T_{R2}) and three at the sender (T_{S1}, T_{S2}, and T_{S3}). T_{S1} maintains the connection as long as there exists a sequence number whose lifetime that has not yet expired; this satisfies **C3**. To satisfy **G1**, T_{S2} prevents the reuse of a sequence number by suspending the transmission of a message with sequence number *x* (from wrap around) if an existing message with sequence number *x* has not been acknowledged.

To partially satisfy **O1**, **O2**, and **C1**, T_{R1} maintains the connection as long as there is an ongoing transmission from the sender (it waits enough time for another message or retransmission to have arrived). To fully satisfy **O1**, **O2**, and **C1** and safely reuse CIDs, T_{S3} maintains the connection until the receiver has released it ($T_{S3} \geq T_{R1}$). T_{R2} handles the safe reuse of a CID by maintaining the connection until the sender is no longer waiting for a sequence number to expire ($T_{R2} \geq T_{S1}$). When the sender loses state, it must also not reuse a CID until the T_{S3} would have expired. Thus, to satisfy **G3**, Feldmeier proposes using stable storage of the

lifetime of each connection or the maximum lifetime that no existing connection exceeds. If the receiver loses state, it must not accept duplicate packets. Stable storage of the highest expiration time of any message will prevent the receiver from accepting duplicates. **G2** is satisfied through error detection and correction.

TP++ performs ARQ error correction and uses selective acknowledgments. An additional feature of TP++ is Forward Error Correction (FEC) codes. These FEC codes reduce retransmission latency by providing enough information to correct packet drops. TP++ segments the TPDU into (Forward Error Correction Blocks) FECBs. TP++ computes a FEC parity code by "striping" across the FECBs. The receiver then has enough information to reconstruct a TPDU if one of the FECBs is lost. Both Delta-t and TP++ improve upon TCP (and T/TCP) by providing a transaction-oriented transport service with minimal RTTs. They address **OBJ1** (transaction-oriented and RTTs are the same for random access) and **OBJ3** (server maintains timer(s)). TP++'s assumption of a synchronized clock and network layer congestion control makes implementation rather difficult over the heterogeneous Internet. As described earlier, the window-based flow control of Delta-t, TCP, and T/TCP will not provide the user with acceptable performance as transmissions start and stop in accordance with the demands of the flow control window. The rate-based flow control of the following transaction-oriented protocols prevent overrunning a host or any intermediate network nodes by estimating and adjusting the time between packet transmissions (the interpacket gap time).

Congestion Avoidance

VMTP and NETBLT are transport protocols with similar features to those described earlier but they provide effective congestion avoidance for high-speed connections using rate-based flow control to satisfy (**OBJ2**).

VMTP

The Versatile Message Transaction Protocol (VMTP) (Cheriton) is a transaction-oriented (client/server) transport protocol designed for quick responses with small amounts of data. VMTP's design originated from a need for efficient Remote Procedure Calls (RPC). VMTP, while providing simple rate-based flow control and negative acknowledgments, also requires both an integrated management module for providing information and notifications and host address-independent naming for what are termed network-visible entities. By sending a request to a server, a client initiates the transaction. The response from the server then terminates the connection. The server does not need to hold any state about the client between transactions. VMTP provides a minimal two packet exchange for short simple transactions and streaming of multi-packet requests and responses. Each message is segmented into one or more packet groups that contain up to a maximum of 32 packets. VMTP groups these packets for acknowledgment, sequencing, selective retransmission, and rate control.

A client uses a monotonically increasing 32-bit transaction number to identify each request. The response to the request will carry the transaction identifier of the request. Consecutive transaction identifiers between packet groups are used as sequence numbers for error control. This transaction identifier is incremented at the end of each message transaction. The transaction identifier is also used by the server to suppress duplicate transaction requests. A server maintains a state record for each client for which it is processing a request and discards

requests with duplicate transaction identifiers (satisfying **O2** and **C1**). To satisfy **G1** and **O1**, VMTP normally retains this record for a period of time to allow the server to filter out any duplicate requests whose time-to-live has not yet expired. To satisfy **G3** and **O1**, if a request arrives and the server does not currently have a state record for the client, the server will perform one of the following:

1. The server may send a probe request to the client to determine the client's current transaction identifier.
2. If the maximum lifetime of packets in the network (plus the maximum VMTP retransmission time) has expired for a previous connection, the server determines that the request is valid.
3. If the operation is specified as idempotent (can be safely redone), the server will simply send the response. The client must then determine the usefulness of the response.

VMTP also provides reliable sequenced transfer of request and response messages through error detection (satisfying **G2**), positive acknowledgment of messages, and timeout and retransmission of lost packets. The response to a request is normally the acknowledgment for a request. The response is acknowledged by either an explicit acknowledgment or another request. An unacknowledged request or response is retransmitted periodically up to some maximum number of retransmissions until it is acknowledged. If a response is idempotent, the response is neither retransmitted nor stored for retransmission—the client must retransmit the request to get the response retransmitted. In other cases, the server maintains a copy of the response until the response is acknowledged or the connection times out.

When a request is sent to a server, each client initializes a timer with the expected time of arrival of the response. If the timer expires before a response arrives, the request is retransmitted. Upon receiving the first packet of a multi-packet response, the client resets the

timer based upon the expected arrival time of the next packet in the response packet group. This timer satisfies **C2** when the client is the receiver and **C3** (the acknowledgment is the response) when the client is the sender.

Each server also maintains a timer for each client whose request is active. The timer determines how long the server waits before timing out between subsequent request packets within a packet group (satisfies **C2** when the server is the receiver). It also determines how long the server waits after sending a response before the server deletes the client's state record (if the last response packet has been transmitted). This timer also satisfies **C3** when the server is the sender.

VMTP uses rate-based flow control to prevent overrunning a host or any intermediate network nodes. By estimating and adjusting the time between packet transmissions (the interpacket gap time), the sender controls the flow of packets to the receiver through intermediate nodes. When the receiver transmits a selective retransmission, the sender may have to increase the interval because packets are being dropped by an intermediate gateway or bridge or the server has been overrun. A sender may use a conservative policy and increase the interpacket gap whenever a packet is lost as part of a multi-packet packet group. With selective retransmissions, the recovery cost to retransmit dropped packets when the packet transmission exceeds the rate of the channel or the receiver is typically less than retransmitting from the first dropped packet. The interpacket gap is expressed in 1/32nds of the Maximum Transmission Unit (MTU) packet transmission time with the range 0 to 8 packet times.

NETBLT

The Network Block Transfer Protocol (NETBLT) (Clark "NETBLT") is another example of a protocol that uses rate control. NETBLT is a unidirectional transport protocol for transmitting large buffers that uses a two-way handshake to setup a connection. A two-way handshake (OPEN and RESPONSE packets) is necessary to negotiate NETBLT's rate-based flow control parameters. NETBLT uses timers to recover from lost OPEN and RESPONSE packets and to determine how long the connection is maintained without a termination message (satisfying **C2** and **C3**). To prevent duplication of OPEN and RESPONSE packets, the OPEN packet contains a unique 32-bit connection ID that is also transferred in the RESPONSE packet (satisfying **G1, G3, O1, O2,** and **C1**). Thus, the sender will not confuse the response to the current request with a response from an earlier incarnation of the connection.

NETBLT breaks each buffer up into DATA packets that are then sent using a datagram service. The end of the buffer is indicated by a LDATA packet. If the LDATA packet is lost, the receiver will not know when the end of the buffer has been transmitted. The expiration of a data timer at the receiver, which is reset when the first DATA packet arrives, indicates the end of transmission if the LDATA packet is lost. Retransmission takes place upon completion of the transfer using a RESEND message in a CONTROL packet containing all the missing packets. Another timer is used by the receiver to handle missing CONTROL messages. When this control timer expires, the receiver resends the control message and resets the timer. After a predetermined number of resends, the receiver assumes that the sender has died and resets the connection. Error detection with checksums satisfies **G2**.

David Clark ("NETBLT") describes how flow control using windows will result in low throughput because the window must be kept small to avoid overflowing hosts and gateways.

Updating this window also requires an end-to-end exchange of messages. With NETBLT's rate control, the transmission rate is negotiated by the senders and receivers during connection setup and after the transmission of a burst of packets. The sender uses timers, rather than messages from the receiver, to maintain the negotiated rate. The sender transmits a burst of packets over a negotiated time interval and then sends another burst. The average transmission time per packet is determined by $\frac{burst\ size}{burst\ rate}$. These parameters are based on packet transmission speed, processing speed, available buffer space, and capacities of the gateways and networks used for the transmission. Each host makes a best guess and tunes the values with subsequent transfers. Clark outlines initial values of five to ten packets (the packet size should be small enough to avoid network layer fragmentation and large enough to limit packet overhead) for the burst size and 60 to 100 milliseconds for the burst rate. Both NETBLT and VMTP use rate-based flow control to prevent the sender from overwhelming the receiver as well as the network and any intermediate nodes. The advantage of rate-based flow control is higher performance without the end-to-end exchange of messages. The key difficulty is determining the appropriate burst size and data transmission rates.

Minimization of Server Load and Server Processing Time

Each connection between a client's browser and a HTTP server requires resources at both the application (HTTP) and transport layers. At the transport layer (regardless of the protocol utilized), the server must maintain the connection information for sequencing, flow control, and packet buffers to provide a reliable transfer of both the HTTP request and the response. The HTTP server must parse and process each request delivered by the transport layer, retrieve or generate the response, and pass a buffer with response onto the transport layer

for delivery to the client's browser. If multiple users request the same object, the transport layer must maintain a separate transmission buffer for each connection containing the same object response. Additionally, the transport layer must segment the response into packets sized for transmission through the network layer.

This research proposes a persistent packet cache to minimize the processing and buffering requirements of both the transport and application layers. Instead of the HTTP server transferring every response to the transport layer, the transport layer will maintain a cache of responses pre-segmented into packets that meet the requirements for Internet hosts. If a transaction desires a larger packet size for efficiency (a local Ethernet transaction, for example), a small buffer may be kept which maps a packet's multiple contents into the cache. After parsing the request, the modified HTTP server will query the transport layer if the response for the object being requested is currently in the cache and is the most updated version (based on the original response file and cache timestamps.) If the cache does not contain the latest version of the response, the HTTP server passes the response onto the transport layer which then caches it. If the transport layer uses rate-based flow control as discussed above to satisfy **OBJ2**, the transport protocol does not rely on the acknowledgment of each octet transmitted and thus, does not have to maintain a window of unacknowledged octets. The transport layer then may forgo a transmission buffer and transmit the response packets (with an appropriate header added) directly from the cache. The sequencing of the packets will reflect their position within the cache. If the transport protocol uses negative acknowledgments exclusively, a NACK will indicate the damaged or lost packet and its position in the cache. Overall, this persistent cache minimizes the resources required by both the transport layer and the HTTP server.

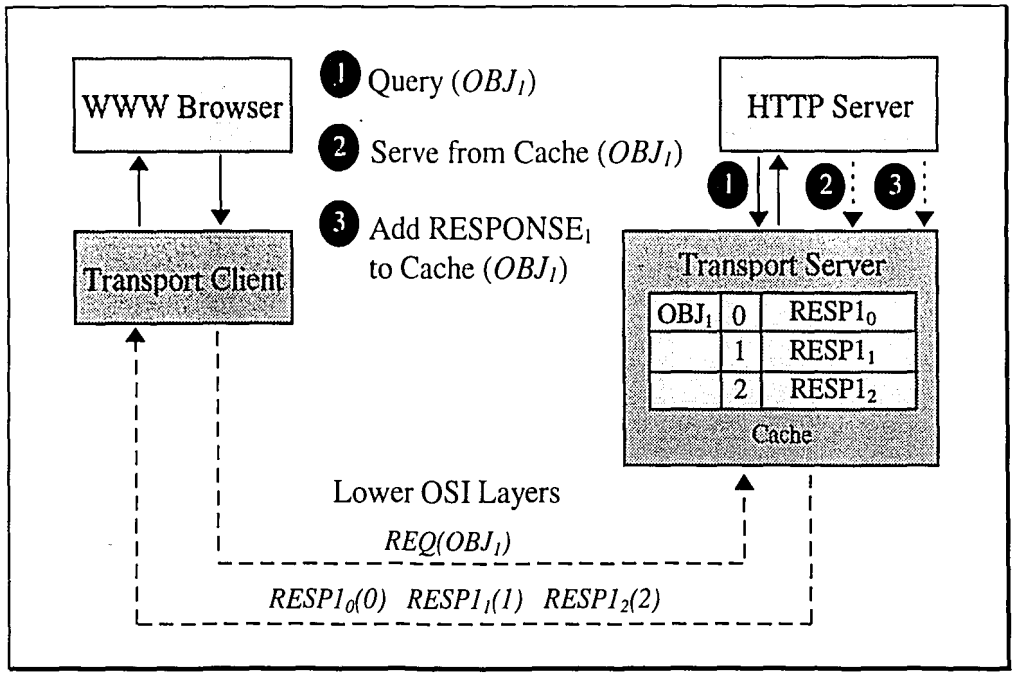

Figure 6: HTTP Request with Packet Cache

As shown in *Figure 6*, after the HTTP request for OBJ_1 is delivered by the transport layer to the HTTP server, the HTTP server determines the object(s) being requested. The HTTP server then queries the transport protocol (1) if the latest response (a three packet response: $RESP1_0$, $RESP1_1$, and $RESP1_2$) for the object(s) is cached. If the cache contains the latest response, the transport protocol delivers the response directly from the cache (2). Otherwise, HTTP adds/replaces the response in the transport protocol's response cache (3) before the transport protocol can deliver it. As shown in *Figure 7*, a NACK indicates the lost packet $RESP1_1$ and its position in the cache (1). *Figure 8* illustrates two client browsers requesting the same object (OBJ_1) while another client requests a different object (OBJ_2).

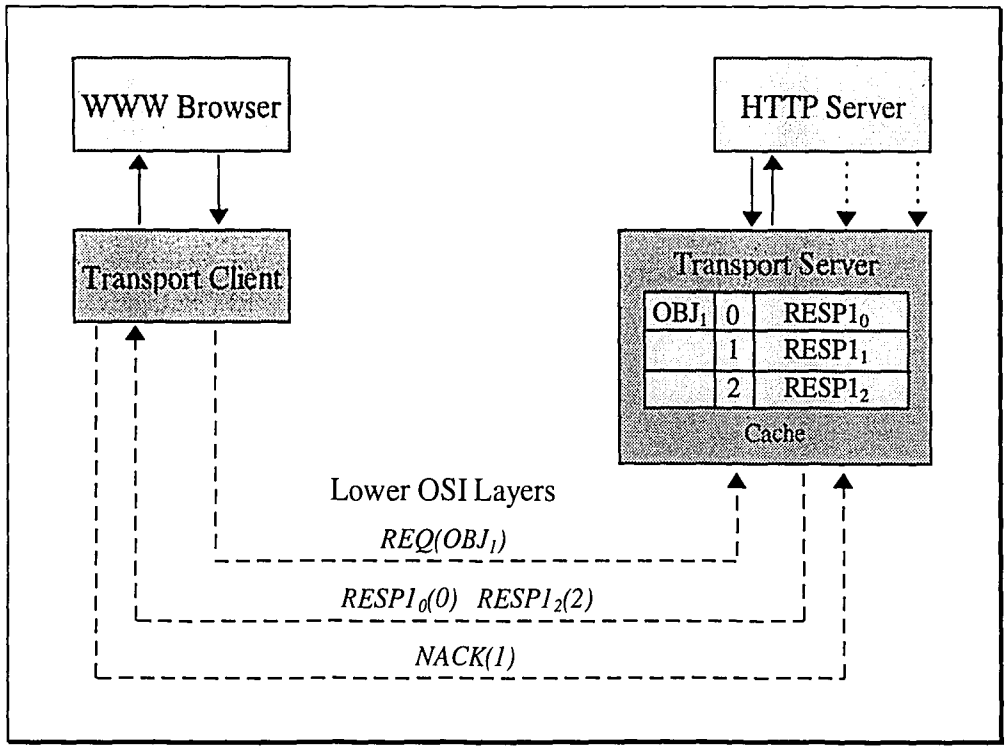

Figure 7: NACK with Packet Cache

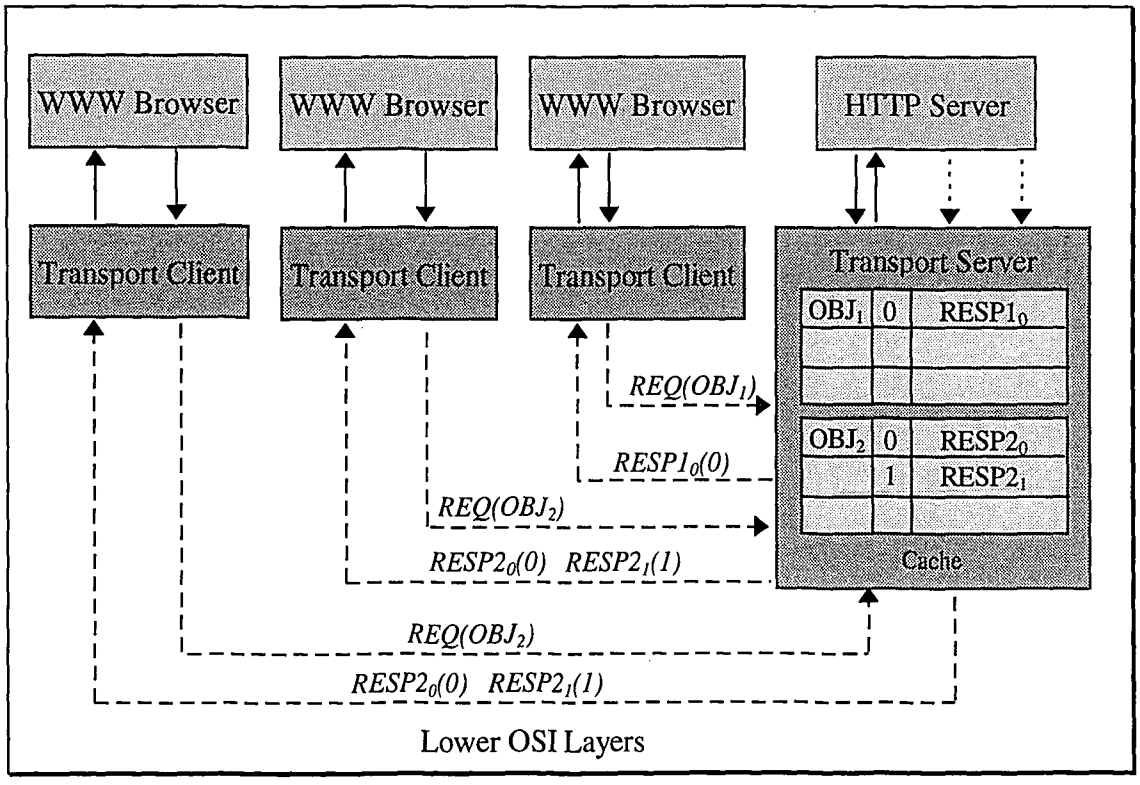

Figure 8: Multiple Browsers with Packet Cache

Object Delivery Transport Protocol

Through the analysis above, this research established the deficiencies of the current HTTP/TCP interaction, demonstrated how existing solutions do not fully address the needs of the World-Wide Web, and established the three objectives for an efficient transport protocol for the WWW.

The final contribution of this research is in the specification of a new transport protocol that addresses **OBJ1**, **OBJ2**, and **OBJ3**. This new protocol called the Object Delivery Transport Protocol (ODTP) utilizes the naming (port and address) from TCP, timer-based connection management from Delta-t and Fletcher, rate-based flow control from VMTP, and the server response cache proposed above. The Object Delivery Transport Protocol (ODTP) provides connection-oriented, transport services for message transactions between a client and server on the WWW. VMTP's model provides the core transport mechanisms of ODTP with its rate-based flow control and negative acknowledgments. ODTP, however, removes the overhead of the explicit management module of VMTP and provides only a simple acknowledgment mechanism to verify that a server has received the request. ODTP also uses the naming of TCP with ports and Internet address. ODTP also differs from VMTP through its use of group sequence numbers (and the *GroupMask*) to identify the position of the packet within the response cache. Like TCP, ODTP uses the network services of a simple (and possibly unreliable) datagram protocol such as IP. To satisfy the connection management requirements outlined above, ODTP relies exclusively on the timer mechanisms of Delta-t and Fletcher. ODTP specifies the following features to address all the objectives above:

1. ODTP is a transaction oriented transport protocol model after VMTP that uses timers exclusively to handle duplicate data detection and minimize the connection management overhead on connection establishment. The minimum transaction latency for ODTP is one RTT plus SPT for random object access regardless of prior readership.
2. ODTP like VMTP incorporates rate-based flow control for effective congestion avoidance for high-speed transmission rates and short connections.
3. Through a response packet cache, ODTP places minimal responsibility on the HTTP and ODTP servers for handling the requests. The cache enables ODTP servers to forgo a transmission buffer in certain situations and send directly from the cache and also access the cache directly for retransmissions.

An overview of the protocol is given below. A proof of the correctness of the protocol's transaction (connection) management and its formal description (including the necessary modifications to the HTTP server) follow. The proof of correctness verifies that while providing the minimum latency of one RTT + SPT, the protocol's transaction management results in its correct operation. Finally, the protocol is analyzed for its ability to address the objectives.

Overview

Following VMTP's model, the ODTP client begins a transaction by creating a unique transaction identifier (*TI*) and segmenting its request into a fixed number of packets groups with up to 32 packets per group. The ODTP client then sends the request to a server. To ensure reliability, ODTP sequences and verifies the checksums of the packets within the packet groups. The server responds to any missing or damaged packets within each group in the request by sending a negative acknowledgment. After a specified interval, if the packet has still not arrived undamaged, these negative acknowledgments are repeated up to a maximum number of times before the transaction fails. These negative acknowledgments also update the rate-based flow control. The server responds to the request by sending the response. If the client does not receive

the response within a specified interval, it sends a message querying the server if it is still processing the request. After a maximum number of queries, the transaction fails. Upon successful reception of the response, the client transfers the message to the application layer.

Transaction (Connection) Management

A transaction is identified by the tuple (*Internet Host Address, Port Identifier, TI*). Each message carries the transaction identifier, source port, and destination port. A transaction is active at the client while it sends the requests, waits for a response from the server, and receives the response. While receiving and servicing a request, a server will not accept any messages from the (*Host, Port*) with a *TI* different from the initial request's *TI* (**O2**). Like Delta-t, timers in ODTP ensure that the client and server maintain a transaction record with the *TI* while packets (including duplicates) with that *TI* exist in the network. Thus, if no transaction exists and the server is willing to receive, no duplicate packets from a previously closed incarnation of a transaction will cause a new transaction to be established and duplicate data to be accepted (**O1**), and no packets from previous transactions will cause an existing transaction to close (**C1**). The client and server will also maintain the transaction record long enough for transmissions, retransmissions, *NACK*s, and *ACK*s to have arrived at the destination (if they are ever going to arrive). Thus, the receiver will not close until it has received all of a sender's possible retransmissions (**C2**). The client will not close until it has received an explicit acknowledgment of all that it has sent in the form of the response or after a fixed number of requests for *ACK*, the expected time of arrival of the response has expired. The server will not close until it has received an additional request from the client or sufficient time has passed for any final *NACK*s to have arrived (**C3**).

To effectively handle these duplicate HTTP responses and packet (re)transmissions with minimal overhead and also significantly enhance the performance of HTTP, the ODTP server maintains a cache of the transaction response packets. The cache contains the response divided into 512 octet segments. The implications of various caching policies is reserved for future research.

For transmission, ODTP segments a message into packet groups that consist of up to 32 individual packets. A packet consists of a 24 octet header and segment data in increments of 512 octets (see *Figure 9*).

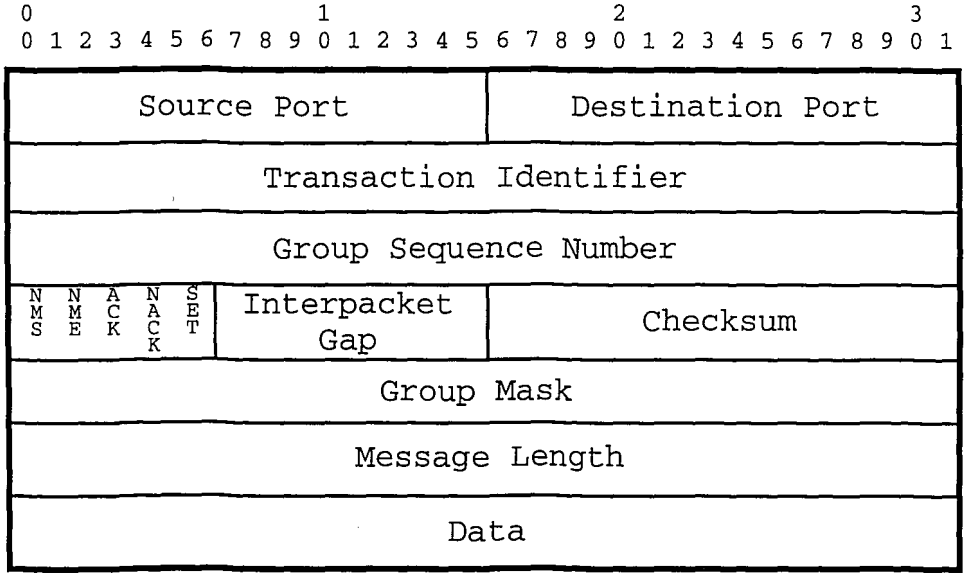

Figure 9: ODTP Packet Format

Following VMTP's design, each packet may contain up to 32 data segments—the entire packet group. The packet size will be determined by the underlying protocols Maximum Transmission Unit (MTU). A 32-bit field in the header call *GroupMask* indicates which data segments (or members of the packet group) are present in the packet with each bit representing one data segment of the 32 possible data segments. If the message is smaller than the packet group's maximum size, the remaining bits are set in the field. If the message does not fit in only one

packet group, the sender utilizes group sequence numbers (*GSN*) on each packet group to enforce the order of the segments. The sender also utilizes two bit fields in the packet header, Not Message Start (*NMS*) and Not Message End (*NME*). The first packet group's header will have the *NMS* bit cleared and the *NME* bit set. The last packet group's header will have the *NME* bit cleared and the *NMS* bit set. Any packet groups sent in between will have both bits set. Thus, a single packet group that is both the start and end of a message will have both bits cleared. After preparing the packet group(s), a sender will transmit the packets within each group at a rate to avoid congesting the network or overwhelming the receiver. A default Interpacket Gap (IPG) is initially used but the receiver may send a packet with the *SET* bit set and indicate an appropriate IPG in the *IPG* field in the packet header.

The first packet received by a server will begin the transaction. The server will create a transaction record (*TR*) for the port and source and *TI* in the packet header and initialize a 32-bit *DeliveryMask* for the packet group to 0. This mask denotes the members of the packet group that have not yet been received. The sender then modifies the *DeliveryMask* to indicate the data segments in the received packet. The sever also checks if there is more than one packet in the message by examining the bit fields for *NMS* and *NME*. If packets are still outstanding (the *DeliveryMask* is not 1 or there are additional packet groups remaining—packets with the *NME* flag cleared have not arrived), the server sets a reception timer to record when it expects the next packet to arrive. If a packet within a packet group is received out of order, the server simply places it in the correct position in the buffer based on the bit position in the *GroupMask* or if it belongs to a different packet group, it is queued until the current packet group is complete. If the reception timer expires, the server transmits a packet with the *NACK* bit set and the *GSN* and

GroupMask set to the expected packet. Subsequent packets update the *DeliveryMask*. Reception is complete when the *DeliveryMask* is 1 and no additional packet groups remain. The server then passes the message onto the application layer. The response is sent using the same methods. A client waits a fixed time interval (defined below) for the response to arrive before requesting an acknowledgment of the request from the server (a packet with the *ACK* bit set). This request is repeated a maximum number of times until either the response to the request or the response to the *ACK* arrives.

Reliability

ODTP provides reliable sequenced delivery of both the request and the response by using sequencing with the *GSN* and the *DeliveryMask* bits (discussed above), transaction identifiers, checksums, and negative acknowledgment and timeout and retransmission of missing packets.

Transaction Identifiers

The ODTP protocol does not restrict the number of incarnations of the same pair (*Internet Host Address*, *Port Identifier*). All packets (request, response, and retransmissions) that are part of the same transaction contain the same *TI*. To satisfy **G3**, **O2**, and **C1**, and prevent duplicate packets from a previous incarnation from affecting the existing transaction (even if an ODTP host crashes), each transaction must have a *TI* that does not already exist in the network from a previous incarnation. Therefore, the initial *TI* on startup (due to a system reset or crash) must be created in the same manner as TCP's initial sequence number. Subsequent *TI*s may be obtained by performing

$$(TI+1) \bmod |TI|. \tag{19}$$

The generation of the initial *TI* is bounded to a 32-bit clock whose low order bit is incremented roughly every 4 microseconds. As a result, the initial *TI* will cycle every 4.77 hours. A proof (**G3**) is given below. Thus, if the MSL is assumed to be the upper bound on a packet's life in the network, the initial *TI* chosen on startup (due to a system reset or crash) will be unique as long as the MSL is less than 4.77 hours. This research assumes that start-up time of a host after a crash is sufficient to ensure any packets that could have the initial *TI* (from a previous connection) have expired.

For a packet from a previous incarnation of a connection to affect an existing connection, the packet must contain the identical transaction identifier. Hence, to satisfy **O2** and **C1**, *TI*s may cycle at the following maximum rate,

$$R_{TI\_MAX} < \frac{|TI|}{MSL} \tag{20}$$

This upper bound is established below (**O2, C1**).

To satisfy **G1** and prevent the reuse of an Group Sequence Number (and its *GroupMask*) while the same GSN (and *GroupMask*) still exists in the network, the maximum rate of creation of the Group Sequence Numbers must be

$$R_{GSN\_MAX} < \frac{|GSN|}{MSL} \tag{21}$$

This upper bound is established below (**G1**).

Checksum

Damage is handled by adding a 16-bit checksum (identical to TCP) to the entire packet transmitted, checking it at the receiver, and discarding damaged segments.

Acknowledgments, Retransmissions, and Timers

ODTP, like TCP, uses the communication services of a simple (and possible unreliable) datagram protocol like the Internet Protocol. IP provides ODTP with the ability to send and receive variable-length segments through multiple heterogeneous networks and interconnecting gateways. Thus, to guarantee the reliability of ODTP's transport services, the protocol must provide mechanisms for retransmitting missing or damaged packets. The client must also receive notification that its request was successfully received so it does not unnecessarily send a duplicate request. The response itself provides this notification.

Each transaction record of a client has a transmission response timer (TC_1). TC_1 is the estimated time to receive a response from the server after transmitting its final request or retransmission packet. This timer prevents a client from waiting for a response that will never arrive due to a busy server or network failure. To guarantee the client receives all the server's *NACK*s, the client must wait upon ending its transmissions and retransmissions for the server to utilize the maximum *NACK* interval to request retransmission of a damaged or lost packet (when the server's reception timer, TS_2 described below, has expired). The client must also wait for the server to process the request and receive the first response packet. Thus, the client must wait the maximum of these two intervals. If all the propagation delays are known ahead of time, TC_1 is initialized upon transmission of the last packet to

$$\delta_{req} + max(\delta_{resp} + SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})), \tag{22}$$

where δ_{req} is the propagation delay of the request (or retransmission), δ_{resp} is the propagation delay of the first response packet to arrive at the client, and $MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})$ is the time to wait for any final *NACK* requests to arrive. Because the propagation delays are not

known ahead of time, an estimate of the RTT is determined and utilized in the timer. TC_1 does not include any waiting time for duplicates—this delay is reserved for the lifetime timer (TC_3). TC_1 is reset to (22) upon receiving a *NACK* and cleared upon reception of the first response packet.

Following VMTP closely, while TC_1 may expire due to a poor estimate of the server's processing time, the cost of an additional acknowledgment when the server receives the response is significant for short transactions. If TC_1 expires before the response is received, an *ACK* request packet is transmitted with the *ACK* bit set. If the server is still processing the request when the *ACK* request arrives, the server replies with an *ACK* to inform the client that its estimate of *SPT* is incorrect. If the server has finished processing the original request, the *ACK* request is discarded. If the client does not receive the response or the *ACK* within an estimate of the $RTT\ (=\delta_{req} + \delta_{resp})$, the *ACK* request is transmitted again. This procedure is repeated for a maximum number of retransmissions. If unsuccessful, the transaction request fails. Thus, a request is acknowledged by the client receiving the response from the server.

Each transaction record of a client also has a reception timer (TC_2) that is used with multi-packet responses. TC_2 is initialized upon reception of the first packet in a multi-packet response to

$$IPG_{max}, \tag{23}$$

where IPG_{max} is the maximum interpacket gap or the maximum amount of time between packet transmissions. TC_2 is reset to (23) upon the successful reception of each packet. If the timer expires before receiving the next packet in the response, a negative acknowledgment is transmitted to the server indicating the packet group and expected packet number. A *NACK* for

the same packet is repeated a fixed number of times ($MaxRetries_{NACK}$) before the transaction fails.

Each transaction record of a client has a lifetime timer (TC_3) that is used to maintain the transaction record as long as the client is known to be sending a request, waiting for a response, receiving a response, or waiting for packets with the *TI* to expire in the network. The transaction itself, though, may be completed before the transaction record is released. Initially, the assumption is made that no duplicates are created within the network or more precisely, all the packets arrive within a known propagation delay. Hence, when transmitting a request and then waiting for the response, TC_3 is the time spent on transmissions, retransmissions, TC_1, and the time spent on *ACK* requests. TC_3 is updated as the transmission progresses as shown below

$$TC_3 = C_{TI\_LTs\_client} = \sum_{i=2}^{n} IPG_i + T + TC_1 + Retries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp}), \tag{24}$$

where n is the number of packets in the request, IPG_i is the interpacket gap for the ith packet in the request (the first packet is assumed to not require an interpacket gap), $Retries_{ACKreq}$ is the number of *ACK* requests transmitted, δ_{ack} is the propagation delay of the *ACK* request, δ_{ack_resp} is the propagation delay of the *ACK* response, and T is the time spent on retransmissions

$$\sum_{j=1}^{r} IPG_{remaining\_j} + \gamma_j, \tag{25}$$

where r is the number of packets retransmitted, $IPG_{remaining\_j}$ is the time remaining in the interpacket gap when the jth *NACK* arrives, and γ_j is the time between the last packet transmission and the reception of the jth *NACK*. For example, if the *NACK* arrives during the transmission of the message, γ_j is simply the time that has expired already in the *IPG*. Thus,

$$IPG_{remaining\_j} + \gamma_j = IPG_k. \tag{26}$$

where IPG_k is the interpacket gap of the next packet to transmit. If the *NACK* arrives when the client is waiting for the response, γ_j resets the wait time interval to the time that had already elapsed. The proof (Lemma 5) (67) given below extends (24) to include the waiting time necessary to receive all duplicates. Initially this waiting time is represented by a factor K. Hence, the lifetime of the transaction record when the client is the sender, C_{TI_LTs}, is

$$TC_3 = C_{TI\_LTs} = \sum_{i=2}^{n} IPG_i + T + \delta_{req} + max(\delta_{resp} + SPT,$$

$$MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})) + Retries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp}) + K = C_{TI\_LTs\_client} + K, \tag{27}$$

where K is the waiting time necessary to ensure all packets (including duplicates) transmitted during the request from either the client or the server (*NACK*s) have arrived at their destination if they are ever going to arrive as shown in (67).

As soon as the client receives a response packet, assuming $S_{TI\_LTs\_server}$ is known by the client, the timer is reset to

$$TC_3 = C_{TI\_LTr} \geq S_{TI\_LTs\_server} - L + K' \tag{28}$$

where $S_{TI\_LT\_server}$ is the server's lifetime timer when transmitting the response (32) when all packets arrive within a known propagation delay, L is the time the server waits to receive any final *NACK*s from the client (the last term of (32)), and K' is the waiting time necessary to ensure all packets transmitted during the response from either the client (*NACK*s) or the server (including duplicates) have arrived at their destination if they are ever going to arrive as shown in

(64). $S_{TI\_LTs\_server}$, however, is not known by the client so an upper bound, $S_{TI\_LTs\_max\_dup}$, (62) which includes K' is developed in Lemma 4.

Each client transaction record on the server has a transaction timer (TS_1). If TS_1 expires before the response is sent, the server assumes the transaction has failed at the client. TS_1 then does not include the propagation delay or transmission of the response. If all propagation delays are known ahead of time, TS_1 is initialized upon successfully receiving the request to

$$max(SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})) + MaxRetries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp}), \tag{29}$$

where δ_{ack} is the propagation delay of the *ACK* request, δ_{ack_resp} is the propagation delay of the *ACK* response, and $MaxRetries_{ACKreq}$ is the maximum number of *ACK* requests. TS_1 is cleared upon transmission of a response packet. Because the propagation delays are not known ahead of time, an estimate of the RTT is determined.

Each client transaction record on the server has a reception timer (TS_2) that is used with multi-packet requests. TS_2 is initialized upon reception of the first packet in a multi-packet request to

$$IPG_{max}, \tag{30}$$

where IPG_{max} is the maximum interpacket gap or the maximum amount of time between packet transmissions by the client. TS_2 is reset to (30) upon the successful receipt of each packet. If the timer expires before the next packet in the request arrives, a negative acknowledgment is transmitted to the client indicating the packet group and expected packet number.

Each client transaction record on the server has a lifetime timer (TS_3) that is used to maintain the transaction record as long as the client is known to be receiving, processing, sending

the response, or waiting for packets with the *TI* to expire in the network. As soon as the server receives a request packet, assuming $C_{TI\_LTs\_client}$ is known by the server, the timer is reset to

$$TS_3 = S_{TI\_LTr} \geq C_{TI\_LTs\_client} - L + K. \tag{31}$$

where $C_{TI\_LTs\_client}$ is the client's lifetime timer while transmitting and waiting for the response (24) when all packets arrive within a known propagation delay, L is the time the client waits to receive any final *NACK*s from the server, and K is the waiting time necessary to ensure all packets transmitted during the request from either the client or the server (*NACK*s) (including duplicates) have arrived at their destination if they are ever going to arrive as shown in (61). $C_{TI\_LTs\_client}$, however, is not known by the server so an upper bound, $C_{TI\_LTs\_max\_dup}$, (59) which includes K is developed in Lemma 4. When ready to transmit the response, TS_3 is reset as described below.

When transmitting the response, TS_3 is the time spent on transmissions, retransmissions, and the time spent waiting for any final *NACKs*. TS_3 is updated as the transmission progresses as shown below

$$TS_3 = S_{TI\_LTs\_server} = \sum_{i=2}^{m} IPG_i + \delta_{resp} + T + MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans}) \tag{32}$$

where m is the number of packets in the response, IPG_i is the interpacket gap for the ith packet in the response or *NACK*, δ_{resp} is the propagation delay of the response, $MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans})$ is the time to wait for any final *NACKs* request to arrive, and T is the time spent on retransmissions

$$\sum_{j=1}^{r} IPG_{remaining\_j} + \gamma_j \tag{33}$$

where r is the number of packets retransmitted, $IPG_{remaining\_j}$ is the time remaining in the interpacket gap when the jth *NACK* arrives, and γ_j is the time between the last packet transmission and the reception of the jth *NACK*. The proof (Lemma 5) (69) given below extends (32) to include the waiting time necessary to receive all duplicates. Initially this waiting time is represented by a factor K'. Hence, the lifetime of the transaction record when the server is the sender, S_{TI_LTs}, is

$$TS_3 = S_{TI\_LTs} = \sum_{i=2}^{m} IPG_i + \delta_{resp} + T + MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans}) + K' =.$$

$$S_{TI\_LTs\_server} + K' \tag{34}$$

where K' is the waiting time necessary to ensure all packets (including duplicates) transmitted during the response from either the client (*NACK*s) or the server have arrived at their destination if they are ever going to arrive as shown in (70).

Rate Control

ODTP, like VMTP, uses rate-based flow control to maintain the flow of packets without overrunning either the server or any intermediate network nodes. ODTP varies the interpacket gap to modify the transmission rate—this research assumes that a default ("safe") rate can be determined, IPG_{max}. The sending ODTP host uses the reception of negative acknowledgments as an indication that the receiver is either experiencing a buffer overflow or that congestion exists on the network. Without any communication between the network and the transport protocol

about network conditions and transmission rates, rate-based flow control must rely on the *NACK*s arriving and not being held up in queues in the network.

Connection Management Correctness

ODTP uses TCP (Postel "TCP"), VMTP (Cheriton), and the mechanisms of Timer-Based protocols outlined by Fletcher as its models. The correctness of ODTP's connection management can be established by proving the connection management requirements discussed above.

Assumption 1 *The maximum packet lifetime is bounded at MSL.*

Assumption 2 *Possibility with low probability of out-of-sequence arrivals through store-and-forward subnetworks, lost or damaged packets, and creation of duplicates within the network.*

Assumption 3 *The client and server have infinite buffers and processing capacity.*

The following parameters are used:

Parameter 1 *The number of available Transaction Identifiers is* $|TI|$.

Parameter 2 *The number of available Group Sequence Numbers is* $|GSN|$.

Parameter 3 *The minimum Interpacket Gap is* IPG_{min}.

Parameter 4 *The estimate of server processing time is SPT.*

Lemma 1

The minimum time interval between generation of sequence numbers for a consecutive packet group X and Y is bounded by IPG_{min}. *Hence, the maximum rate of generation of GSNs is* $R_{GSN\_max} = \frac{1}{IPG_{min}}$.

Proof: Let n denote the minimum number of packets to send a packet group X. Packets are transmitted with an interpacket gap of $IPG_i \geq IPG_{min}$. Thus, the time interval between generation

of the group sequence number for X and Y is the time to send group X and the time to wait before sending group Y,

$$\sum_{i=2}^{n} IPG_i + IPG_j. \tag{35}$$

If $n=1$, the minimum time interval between generation of group sequence numbers for a consecutive packet group X and Y becomes IPG_j which is bounded by IPG_{min}. If the minimum interval between generation of consecutive group sequence numbers is IPG_{min}, the maximum rate of generation of group sequence number, R_{GSN_max}, is

$$\frac{1}{IPG_{min}}. \tag{36}$$

Theorem 1

ODTP satisfies Transaction (Connection) Management General Requirement 1.

General 1

An identifier of an information unit used for error control must not be reused while one or more copies of that unit or its NACK are alive.

Proof: By Lemma 1, R_{GSN_max} is the maximum rate at which packet groups are generation. Thus, group sequence numbers will wrap around if the *GSN* number space is smaller than the product of the rate creation of the *GSN*s and their lifetimes in the network,

$$|GSN| \leq R_{GSN\_max} \cdot LT, \tag{37}$$

where LT is the lifetime of the identifiers. Recall, a packet has a network LT of MSL. Therefore, to prevent wrap around

$$|GSN| > R_{GSN\_max} \cdot MSL. \tag{38}$$

Thus, because MSL is defined by the network, the choice of $|GSN|$ and R_{GSN_max} must satisfy

$$\frac{|GSN|}{R_{GSN\_max}} > MSL \tag{39}$$

but $|GSN|$ is fixed. Thus,

$$\left(\frac{1}{IPG_{min}} = R_{GSN\_max}\right) < \frac{|GSN|}{MSL} \tag{40}$$

Hence, to prevent an identifier of an information unit used for error control (the group sequence number) from being reused while one or more copies of that unit or its NACK are alive, the minimum interpacket gap must be large enough to satisfy (40).

Lemma 2

The minimum time interval between generation of consecutive TIs at the client is bounded by RTT + SPT. Hence, the minimum lifetime of a TI at the client (C_{TI_LT}) is also RTT + SPT.

Proof: Let δ denote the transfer delay of packets between the client and server. Let n and m denote the number of packets in the request and response, respectively. Thus, with no retransmissions and acknowledgment requests, the transaction at the client takes

$$\sum_{i=2}^{n} IPG_i + \delta_{req} + SPT + \sum_{i=2}^{m} IPG_j + \delta_{resp} \tag{41}$$

where IPG_i is the interpacket gap for the ith packet in the request, IPG_j is the interpacket gap for the jth packet in the response, δ_{resp} is the propagation delay of the response, and δ_{req} is the propagation delay of the request. The client does need to wait for any final *NACK*s as shown in

(22) because the lifetime timer is cleared when the first response packet is received. Any waiting time for duplicates occurs after the transaction is completed. At the minimum n and m of 1, (41) becomes

$$\delta_{req} + \delta_{resp} + SPT \tag{42}$$

and

$$\delta_{req} + \delta_{resp} = RTT, \tag{43}$$

where RTT is an estimate of the round-trip time. Hence,

$$RTT + SPT, \tag{44}$$

is the minimum interval between generation of *TI*s. Equation (44) also indicates the minimum lifetime of the *TI* at the client ($C_{TI\_LT\_min}$).

Lemma 3

The minimum lifetime of the transaction identifier for a client's transaction record at the server is bounded by $S_{TI\_LT\_min} \geq C_{TI\_LT\_min} + MaxRetries_{NACK} \cdot RTT + K'$.

Proof: From (31) and (35) the lifetime of the client's transaction record at the server is the time spent receiving and processing the request and the time spent sending the response,

$$TS_3 \geq C_{TI\_LTs\_client} - L + K + S_{TI\_LTs} \tag{45}$$

where L is the time the client waits to receive any final *NACK*s from the server and K is the waiting time necessary to ensure all packets transmitted during the request from either the server (*NACK*s) or the client (including duplicates) have arrived at their destination if they are ever going to arrive. With no retransmissions, acknowledgment requests, and a minimum n and m of

1 as assumed in Lemma 2 and the reset of the server's timer when the request has completely arrived, the minimum TS_3 becomes

$$S_{TI\_LT\_min} \geq \delta_{req} + SPT + S_{TI\_LTs} \tag{46}$$

From (41) with no retransmissions and m of 1,

$$S_{TI\_LTs} = \delta_{resp} + MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans}) + K' \tag{47}$$

where K' is the waiting time necessary to ensure all packets transmitted during the response from either the client ($NACK$s) or the server (including duplicates) have arrived at their destination if they are ever going to arrive. Hence,

$$S_{TI\_LT\_min} \geq \delta_{req} + SPT + \delta_{resp} + MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans}) + K' \tag{48}$$

From Lemma 2 using (43),

$$S_{TI\_LT\_min} \geq C_{TI\_LT\_min} + MaxRetries_{NACK} \cdot RTT + K'. \tag{49}$$

Theorem 2

ODTP satisfies Transaction (Connection) Management Transaction Opening Requirement 2 and Transaction Closing 1. When an initial Transaction Identifier is chosen for the transaction, it must be such that no duplicates from previous transactions can be accepted within the current transaction. Hence, the rate of creation of the Transaction Identifiers must be bounded by $\frac{|TI|}{MaxRetries_{NACK} \cdot RTT + K'}$.

Transaction Opening 2

If a transaction exists, then no packets from a previously closed transaction should be acceptable within a current connection.

Transaction Closing 1

No packet from a previous transaction should cause an existing transaction to close.

Proof: Assuming there has been no crash with loss of memory, a client will choose the next TI by performing

$$(TI+1) \bmod |TI|. \tag{50}$$

*TI*s, though, suffer from the same wrap around problem as *GSN*s. Thus, *TI*s will wrap around and duplicates from previous transactions with the same *TI* could be accepted within the current transaction if

$$|TI| \leq S_{TI\_LT\_min} \cdot R_{TI\_max}, \tag{51}$$

where $R_{TI\_max} = \dfrac{1}{C_{TI\_LT\_min}}$ is the rate of generation of *TI*s by the client using Lemma 2 and $S_{TI\_LT\_min}$ is the minimum lifetime of the transaction at the server from Lemma 3. To prevent wrap around,

$$|TI| > S_{TI\_LT\_min} \cdot \frac{1}{C_{TI\_LT\_min}}, \tag{52}$$

but $|TN|$ is fixed. By Lemma 3, to prevent wrap around

$$|TI| > (C_{TI\_LT\_min} + MaxRetries_{NACK} \cdot RTT + K') \cdot \frac{1}{C_{TI\_LT\_min}}$$

$$|TI| > \frac{MaxRetries_{NACK} \cdot RTT + K'}{C_{TI\_LT\_min}} \tag{53}$$

Therefore, the *TI*s will not wrap around if the rate of creation of the identifiers by the client (R_{TI_max}) is

$$\left(R_{TI\_max} = \frac{1}{C_{TI\_LT\_min}} \right) < \frac{|TI|}{MaxRetries_{NACK} \cdot RTT + K'}. \tag{54}$$

Theorem 3

ODTP satisfies Transaction (Connection) Management General Requirement 2.

General 2

The error control information must itself be error corrected.

Proof The 16-bit checksum is applied to the entire packet including the *TI*s and *GSN*s.

Lemma 4

To receive all the sender's transmissions, retransmissions, and ACKs including duplicates, the receiver must maintain its transaction record for

$$receiver_{TI\_LT} \geq sender_{TI\_LTs\_max\_dup} - L, \tag{55}$$

where $sender_{TI\_LTs\_max\_dup}$ *is the maximum lifetime of the sender and ensures all packets including duplicates transmitted by the receiver and sender (NACKs) have arrived at their destination if they are ever going to arrive and L is the time the sender waits to receive any final NACKs from the receiver. Thus, while the client is sending, the server must maintain its transaction record (and identifier) for the client for*

$$S_{TI\_LTr} \geq C_{TI\_LTs\_max\_dup} - L,$$
$$S_{TI\_LTr} = (n\text{-}1) \cdot IPG_{max} + T + 2\ MSL + SPT + MaxRetries_{ACKreq} \cdot 2\ MSL.$$

While the server is sending, the client must maintain the transaction record for

$$C_{TI\_LTr} \geq S_{TI\_LTs\_max} - L,$$
$$C_{TI\_LTr} = (m\text{-}1) \cdot IPG_{max} + T + MSL.$$

Proof The transmission of a request with n packets by the client requires at most

$$\sum_{i=2}^{n} IPG_i + T + \delta_{req}, \tag{56}$$

where T is the time spent on retransmissions as shown in Equation (25) and IPG_{max} is the maximum interpacket gap as shown in Equation (23). Recall, the response from the server serves as the acknowledgment of the request. Thus, from (22), the client will wait for the response at most

$$max(\delta_{resp} + SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})),$$

(57)

before requesting an immediate acknowledgment (*ACK* request) at most *MaxRetries* times—to check if the server is down or simply slow, from (24),

$$MaxRetries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp}).$$

(58)

Thus, the lifetime of the transaction record when the client is the sender, C_{TI_LTs}, is (27) as shown earlier.

The server, though, does not have the advanced knowledge necessary to accurately set its lifetime timer when the initial request is received. Recall, with rate-based flow control, the sender does not rely on acknowledges to continue the transmission of packets. Thus, if a link temporarily fails or there is an intermittent delay, the sender will continue the flow of packets to the receiver until all the packets have been transmitted. The receiver, therefore, must maintain the transaction for the duration of the expected transmission. Any missing segments may be requested using *NACK*s. Thus, while the server will update (reduce) its lifetime timer as additional packets are received, it must set an upper bound to ensure any temporary link failures and intermittent delays to do not incorrectly terminate the transaction. If the server knows only the length of the request (the *MessageLength* field in the header), the server will bound IPG_i with IPG_{max} and determine that the client will maintain its transaction record while transmitting the requesting and waiting for the response at most

$$C_{TI\_LTs\_max} = (n\text{-}1) \cdot IPG_{max} + T + \delta_{req} + max(\delta_{resp} + SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})] + MaxRetries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp}))$$

(59)

δ is also not known in advance but is bounded by MSL (a packet or its duplicate will not arrive if $\delta > MSL$) so the lifetime of the client's transaction record including the time for any duplicate packets from the server (*NACK*s) or the client to arrive if they are ever going to is

$$C_{TI\_LTs\_max\_dup} = (n-1) \cdot IPG_{max} + T + MSL + max[MSL + SPT, MaxRetries_{NACK} \cdot 2\ MSL] + MaxRetries_{ACKreq} \cdot 2\ MSL. \tag{60}$$

L is $MaxRetries_{NACK} \cdot 2\ MSL$. Therefore, if the server maintain the client's transaction record for

$$S_{TI\_LTr} \geq C_{TI\_LTs\_max\_dup} - L,$$
$$S_{TI\_LTr} = (n-1) \cdot IPG_{max} + T + 2\ MSL + SPT + MaxRetries_{ACKreq} \cdot 2\ MSL, \tag{61}$$

the server will receive all the client's transmissions, retransmissions, and *ACK*s including duplicates.

Similarly, the lifetime of the transaction record when the server is the sender S_{TI_LTs} is (34). If the client knows only the length of the response (the *MessageLength* field in the header), the client will bound IPG_i with IPG_{max} and determine that the server will maintain its transaction record while transmitting the response at most

$$S_{TI\_LTs\_max} = (m-1) \cdot IPG_{max} + \delta_{resp} + T + MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans}), \tag{62}$$

δ is also not known in advance but is bounded by MSL (a packet or its duplicate will not arrive if $\delta > MSL$) so the lifetime of the server's transaction record including the time for any duplicate packets from the client (*NACK*s) or the server to arrive if they are ever going to is

$$S_{TI\_LTs\_max\_dup} = (m-1) \cdot IPG_{max} + T + MSL + MaxRetries_{NACK} \cdot 2\ MSL. \tag{63}$$

L is $MaxRetries_{NACK} \cdot 2\ MSL$. Therefore, if the client maintains the transaction record for

$$C_{TI\_LTr} \geq S_{TI\_LTs\_max\_dup} - L,$$
$$C_{TI\_LTr} = (m-1) \cdot IPG_{max} + T + MSL, \tag{64}$$

the client will receive all the server's transmissions, retransmissions, *ACK*s, and duplicates.

Lemma 5

To receive all the receiver's NACKs (including duplicates), the sender must maintain its transaction record for

$$sender_{TI\_LT} \geq TT + K, \tag{65}$$

where TT is the maximum transmission time (including propagation delays) and K is the time the sender must wait to receive any final NACKs.

Thus, while the server is sending, the server must maintain its transaction record (and identifier) for the client for

$$S_{TI\_LTs} \geq TT + K',$$

$$S_{TI\_LTs} = \sum_{i=2}^{m} IPG_i + MSL + T + MaxRetries_{NACK} \cdot 2\ MSL.$$

While the client is sending the request, the client must maintain its transaction record for

$$C_{TI\_LTs} \geq TT + K,$$

$$C_{TI\_LTs} = \sum_{i=2}^{n} IPG_i + T + MSL + max(MSL + SPT, MaxRetries_{NACK} \cdot 2\ MSL) +$$

$$Retries_{ACKreq} \cdot 2\ MSL.$$

Proof From (27), the transmission of a request with n packets by the client and the waiting time for the response is

$$C_{TI\_LT\_client} = \sum_{i=2}^{n} IPG_i + T + \delta_{req} + max(\delta_{resp} + SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})) +$$

$$MaxRetries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp}) \tag{66}$$

and δ is bounded by MSL (a packet or its duplicate will not arrive if $\delta > MSL$) so the lifetime of the client's transaction record including the time to receive any *NACK*s and duplicates is

$$C_{TI\_LTs} = \sum_{i=2}^{n} IPG_i + T + MSL + max(MSL + SPT, MaxRetries_{NACK} \cdot 2\ MSL) + Retries_{ACKreq} \cdot 2$$
$$MSL = C_{TI\_LT\_client} + K.$$
(67)

Therefore, if the client maintain its transaction record for

$$C_{TI\_LTs} \geq TT + K,$$
$$C_{TI\_LTs} = \sum_{i=2}^{n} IPG_i + T + MSL + max(MSL + SPT, MaxRetries_{NACK} \cdot 2\ MSL) +$$
$$Retries_{ACKreq} \cdot 2\ MSL,$$
(68)

the client will receive all the server's *NACK*s including duplicates.

From (32), the transmission of a response with m packets by the server and the waiting time for the response is

$$S_{TI\_LT\_server} = \sum_{i=2}^{m} IPG_i + \delta_{resp} + T + MaxRetries_{NACK} \cdot (\delta_{NACK} + \delta_{retrans}),$$
(69)

δ is bounded by MSL so the lifetime of the server's transaction record plus the time to receive any *NACK*s and duplicates is

$$S_{TI\_LTs} = \sum_{i=2}^{m} IPG_i + MSL + T + MaxRetries_{NACK} \cdot 2\ MSL = S_{TI\_LT\_server} + K'$$
(70)

Therefore, if the server maintains its transaction record for

$$S_{TI\_LTs} \geq TT + K',$$
$$S_{TI\_LTs} = \sum_{i=2}^{m} IPG_i + MSL + T + MaxRetries_{NACK} \cdot 2\ MSL.$$
(71)

the server will receive all the client's *NACK*s including duplicates.

Theorem 4

ODTP satisfies Transaction (Connection) Management Transaction Opening Requirement 1, Transaction Closing Requirement 2, and Transaction Closing Requirement 3. Thus, both sides must maintain the transaction long enough so duplicates can be detected, and transmissions, retransmissions, NACKs, and ACKs have arrived at their destination (if they are ever going to arrive).

Transaction Opening 1
If no connection exists and the receiver is willing to receive, no duplicate packets from a previously closed connection should cause a new connection to be established.

Transaction Closing 2
A receiving side should not close until it has received all of a sender's possible retransmissions and can respond to them.

Transaction Closing 3
A sending side should not close until it has received acknowledgment of all that it has sent.

Proof By Lemma 4, the receiver will not release its transaction record until all the sender's packets including duplicates have arrived at the receiver (**C2**). Also from Lemma 4, no duplicate packets from an earlier incarnation of a connection will cause a new connection to be established (**O1**).

By Lemma 5, if the client is the sender, it will maintain the transaction until it either receives the response (the acknowledgment) or the transaction fails and enough time has expired for any *NACK*s (and duplicates) to have been sent and expired in the network. If the server is the sender, it will maintain the connection until enough time has expired for any NACKs (including duplicates) to have been sent and expired in the network. Thus, **C3** is true.

Theorem 5

ODTP satisfies Transaction (Connection) Management General Requirement 3.

General 3
If the crash of an end can cause it to lose its state, then appropriate crash recovery mechanism must assure the other requirements G1, G2, O1, O2, C1, C2, and C3.

Proof If a crash occurs, the client or server may lose their connection state. **G2** will not be affected by a loss of connection state and is satisfied if a crash occurs. To ensure that all the other requirements are met **G1**, **O1**, **O2**, **C1**, **C2**, and **C3**, we must ensure that all packets that currently exist in the communication network or are being transmitting (recall, with ODTP's rate-based flow control, there is no flow of acknowledgments back to the sender.) Thus, the protocol must maintain the transaction until its lifetime would have expired. Hence, some stable storage is required for the transaction records.

Additionally, when a new transaction is initiated, an initial 32-bit Transaction Identifier is generated by using a 32-clock which is possibly fictitious. The low order bit of this 32-bit clock is incremented roughly every 4 microseconds (Postel "TCP"). Thus, the clock values will wrap around after

$$2^{32} \cdot 4\,\mu s = \frac{2^{34}}{10^6 \frac{\mu s}{s}} = \frac{17179.869 s}{3600 \frac{s}{h}} = 4.772 h. \tag{72}$$

Therefore, the initial Transaction Identifiers will cycle approximately after 4.772 hours. If

$$MSL < 4.772\ hours \tag{73}$$

the initial Transaction Identifiers will be unique. Postel ("TCP") specifies the MSL is 2 minutes. Hence, the initial Transaction Identifiers will be unique even after a crash.

Timer Rules

Client Timers	(Re)set Event	Reset/Initial Value	Timer Event
Transmission Timer (TC_1)	[1] Transmission of last request and retransmission packet [2] Transmission of *ACK* request	[1] $\delta_{req} + max(\delta_{resp} + SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK}))$ (22) [2] $\delta_{req} + \delta_{resp}$ *(RTT)*	[1] Expiration: Transmit *ACK* request up to *MaxRetries* times, reset TC_1 each time to [2] [2] If *ACK* request has been sent *MaxRetries* times, transaction failure [3] If ACK response is received, reset TC_1 to [1]
Reception Timer (TC_2)	Reception of a packet	IPG_{max} (23)	Transmit *NACK* with packet group and expected packet number
Lifetime Timer on Transmitting Request and Waiting for Response (TC_3)	Transmission of last request or retransmission packet	$MSL + max(MSL + SPT, MaxRetries_{NACK} \cdot 2\ MSL) + Retries_{ACKreq} \cdot 2\ MSL$ (68)	[1] Expiration: Transaction failure
Lifetime Timer on Receiving Response (TC_3)	Reception of first response packet	$(packets\ remaining\text{-}1) \cdot IPG_{max} + T + MSL$ (64)	[1] Expiration: if transaction completed, transaction success otherwise, failure [2] Updated after reception of each packet:

Figure 10: Client Timer Rules

Server Timers	(Re)set Event	Reset/Initial Value	Timer Event
Transaction Timer (TS_1)	Reception of last request packet	$max(SPT, MaxRetries_{NACK} \cdot (\delta_{retrans} + \delta_{NACK})) + MaxRetries_{ACKreq} \cdot (\delta_{ack} + \delta_{ack\_resp})$ (29)	[1] Transaction failure
Reception Timer (TS_2)	Reception of a packet	IPG_{max} (30)	[1] Transmit *NACK* with packet group and expected packet number
Lifetime Timer on Transmitting Response (TS_3)	Transmission of last response or retransmission packet	$MSL + T + MaxRetries_{NACK} \cdot 2\ MSL$ (70)	[1] Transaction completed
Lifetime Timer on Receiving Request (TS_3)	Reception of first request packet	$(packets\ remaining\text{-}1) \cdot IPG_{max} + T + 2\ MSL + SPT + MaxRetries_{ACKreq} \cdot 2\ MSL$ (61)	[1] Transaction failed

Figure 11: Server Timer Rules

Formal Description

Note: $\oplus$ denotes addition mod $|TI|$ or $|GSN|$
TR is a transaction record
P is a packet

Global Variables for both Client and Server

TI	/* transaction identifier */
IPG	/* interpacket gap [transmission rate] */
IPG_{min}	/* minimum interpacket gap [maximum transmission rate] */
IPG_{max}	/* maximum interpacket gap [minimum transmission rate] */
$IPG_{Adjustment}$	/* interpacket gap adjustment [transmission rate adjustment]*/
SPT	/* estimate of Server Processing Time in milliseconds */
RTT	/* estimate of Round Trip Time in milliseconds */
HostAddr	/* host's Internet address */
MSS	/* maximum segment size */
MSL	/* maximum segment lifetime */
MaxSegsPerGroup	/* maximum segments per packet group */
MaxRetries	/* maximum retransmissions */
HeaderSize	/* size of ODTP packet header */
MTU	/* maximum transmission unit */
MaxRetriesAckReq	/* maximum number of retries for ack */
Cache	/* cache of data segments */

Server

The transaction record TR_s at the server contains the following components for each transaction:

TI	/* transaction identifier */
RTT	/* estimate of Round Trip Time */
SPT	/* estimate of Server Processing Time */
TS_1	/* transmission timer */
TS_2	/* reception timer */
TS_3	/* lifetime */
TS_4	/* data rate timer */
RetransmissionCount	/* retransmission counter (for each packet) */
AckRequestCount	/* ACK request count */
GrpTransmissionMask	/* bit mask indicating the portions of the group transmitted; */ /* set before entering the transmission queue and cleared */ /* incrementally as the 512-octet packets of the group */ /* are transmitted. */
DeliveryMask	/* bit mask indicating the portions of the group received; */ /* cleared as the 512-octet packets of the group are received. */
GSN	/* group sequence number */
TPGB	/* transmission buffer of packet groups and the headers and */ /* cache indices of its members */
RPGB	/* reception buffer of packet groups */

WQ	/* waiting queue */
RTQ	/* retransmission queue of packet groups */
State	/* state = {*Transmitting, Processing, Receiving, Waiting*} */
P	/* a packet */
DirectCache	/* send directly from cache? */
CurrentIndex	/* current index into cache if sending directly from the cache */
PacketCount	/* number of packets received so far */
ServerAddr	/* server's address */
ServerPort	/* server's port */
ClientAddr	/* client's address */
ClientPort	/* client's port */

Initialization:

IPG ← system default interpacket gap
IPG_{min} ← system default minimum interpacket gap
IPG_{max} ← system default maximum interpacket gap
$IPG_{adjustment}$ ← system default interpacket gap adjustment
SPT ← default estimate of server processing time
RTT ← default estimate of round trip time
HostAddr ← host's Internet address
MSS ← *512*
MSL ← *120 s*
MaxSegsPerGroup ← *32*
MaxRetries ← *3*
HeaderSize ← *24*
MTU ← *536* (IP), *1536* (Ethernet)
MaxRetriesAckReq ← *3*
Cache← *0*

Receive Request Packet (*P*)

/* does current source (port and address) and *P.Hdr.TI* already have a *TR* */
if not TR_s *(P.Hdr.ClientAddr, P.Hdr.ClientPort, P.Hdr.TI)* exists
then
Allocate a transaction record TR_s *(srcAddr, srcPort, TI)*
/* initialize variables */
TR_s*.SPT* ← *SPT*
TR_s*.TI* ← *P.Hdr.TI*
TR_s*.RTT* ← *RTT*
TR_s*.IPG* ← *IPG*
TR_s*.ClientAddr*← *P.Hdr.ClientAddr*
TR_s*.ClientPort* ← *P.Hdr.ClientPort*
TR_s*.ServerAddr* ← *HostAddr*
TR_s*.ServerPort*← *port*
TR_s*.AckRequestCount*← *0*

```
        TR_s.PacketCount ← 0
        TR_s.State ← Receiving
        /* === TIMERS === */
        /* transaction timer TS_1 is cleared, reception timer TS_3 is set to expected arrival*/
        TR_s.TS_1 ← ∞
        TR_s.TS_2 ← IPG_max
        /* set lifetime timer to upper bound of client's lifetime based */
        /* on message length and number of packets received so far */
        TR_s.TS_3 ← P.Hdr.MessageLength·IPG_max + 2 MSL +
                    SPT + MaxRetriesAckReq · 2 MSL
        /* data rate timer (for transmission) is cleared */
        TR_s.TS_4 ← ∞
        TR_s.RetransmissionCount ← 0
        TR_s.DeliveryMask ← 0
        TR_s.GSN ← 0
end if
/* Retrieve Transaction Record */
TR ← TR_s (P.Hdr.ClientAddr, P.Hdr.ClientPort, P.Hdr.TI)
if TR.State = Receiving
then
    /* increment count of packets received, update upper bound on */
    /* client's lifetime, reset reception timer, and either place in buffer in the */
    /* correct spot or place in the waiting queue */
    TR.PacketCount ← TR.PacketCount + 1
    TR.TS_3 ← (P.Hdr.MessageLength-TR.PacketCount)·IPG_max + 2 MSL +
              SPT + MaxRetriesAckReq · 2 MSL
    /* reset reception timer */
    TR.TS_2 ← IPG_max
    /* if packet belongs to current group */
    if P.Hdr.GSN = TR.GSN
        then /* add to reception buffer and update delivery mask */
             TR.RPBQ[i,P.Hdr.GroupMask] ← P
             TR.DeliveryMask ← TR.DeliveryMask + P.Hdr.GroupMask
             /* if DeliveryMask is 1 then packet group is complete */
             if TR.DeliveryMask = 1
                 then TR.GSN ← TR.GSN ⊕ 1
                      move any TR.WQ packets with TR.GSN to
                          TR.RPBQ[GSN]
             end if
        /* otherwise out-of-sequence and add to Waiting Queue */
        else add P to TR.WQ
    end if
```

```
        /* if the last packet, process the request */
        if  not P.Hdr.NME and TR.DeliveryMask = 1
            /* clear reception timer TS2, set transaction timer TS1 */
            TR.TS2 ←∞
            TR.TS1 ← max(TR.SPT, MaxRetries · TR.RTT) + MaxRetriesAckReq·TR.RTT
            /* process the request */
            TR.State ← Processing
            cacheIndex ← HTTP Server Process Request (TR.RPBQ, responseMsg)
            /* is element in cache already? if no, cache it*/
            if  cacheIndex < 0
                then  cacheIndex ← Cache Message (responseMsg)
            end if
            TR.CacheIndex ← cacheIndex
            /* done processing, clear transaction timer and lifetime timers */
            TR.TS1 ←∞
            TR.TS3 ←∞
            TR.State ← Transmitting
            TR.RetransmissionCount ← 0
            TR.GrpTransmissionMask ← 0
            TR.GSN ← 0
            send packet groups (TR, responseMsg)
        end if
    end if


Find Cache Index (objects)
    if  objects in Cache
        then
            return objects index
        else
            return -1
    end if

Is Latest Version (objects, timestamp)
    if  objects in Cache
        then
            return objects.timestamp=timestamp
        else
            return FALSE
    end if
```

Cache Message (*msg*)

find free position *i*
use a cache replacement strategy if necessary
split the msg of size *n* bytes (octets) into *m* MSS-sized octet segments
∀j: j={0..m-1}, Cache[i,j] ← msg(j,MSS)

/* store the last segment size with each segment */
Cache[i].LastSegmentSize = size(Cache[i,m-1])

/* store the total number of packets required for the message */
Cache[i].NumPackets = m
return *i*

Send Packet Groups (*TR, msg*)

/* initialize default Packet Header */
TR.P.Hdr.SegmentSize ← 0
TR.P.Hdr.ServerAddr ← TR.ServerAddr
TR.P.Hdr.ServerPort ← TR.ServerPort
TR.P.Hdr.ClientAddr ← TR.ClientAddr
TR.P.Hdr.ClientPort ← TR.ClientPort
TR.P.Hdr.TI ← TR.TI
TR.P.Hdr.GSN ← 0
TR.P.Hdr.Data ← 0
TR.P.Hdr.ACK← 0
TR.P.Hdr.NACK ← 0
TR.P.Hdr.NMS ← 1
TR.P.Hdr.NME ← 1
/* current index into cache—used if transmitting directly from the cache */
TR.CurrentIndex ← 0
/* if MTU is larger than the MSS-sized segments in the cache, create a queue of the */
/* packet headers with multiple mappings into the cache */
Queue packet groups *(TR, msg)*
TR.GSN ← 0
Set Transmission Mask *(TR, TR.GSN)*
Send Packet *(TR, TR.GSN)*

Send Packet Directly From Cache (*TR, index*)

if first packet group
then
TR.P.Hdr.NMS ← 0
TR.P.Hdr.NME ←1
end if

```
    if  last packet group
        then
            TR.P.Hdr.NMS ← 1
            TR.P.Hdr.NME ←0
    end if

/* set packet group mask */
    TR.P.Hdr.GroupMask← 2 index mod (MaxSegsPerGroup-1)
    TR.P.Hdr.Data ← Cache[TR.CacheIndex, index]
    if  last packet in last packet group
        then
            TR.P.Hdr.GroupMask ← 1
    end if

    if  index ← Cache[TR.CacheIndex].NumPackets-1
        then
            TR.P.Hdr.SegmentSize← TR.Cache[TR.CacheIndex, index].LastSegmentSize
        else
            TR.P.Hdr.SegmentSize← MSS
    end if

Queue Packet Groups (TR, msg)
    /* more than one data segment per packet */
    if  MTU>MSS + HeaderSize
        then
            TR.DirectCache = FALSE
            create a queue of m packet groups TR.TPGB[m,j] and their j elements (E) of
            packet headers and cache indices
            each packet may contain more than one cache segment but each packet may
            contain at most one entire packet group [all MaxSegsPerGroup
            segments]
                /* update group mask for segments i = 0.. (MaxSegsPerGroup-1) */
                /* in the packet added, k is the cache index */
                E.Hdr.GroupMask ← E.Hdr.GroupMask + 2^i
                E.Hdr.SegmentSize ← E.Hdr.SegmentSize + (MSS or LastSegmentSize)
                E.Data[i] ← k
                if last packet in last packet group, E.Hdr.GroupMask ← 1
            map a packet's contents to the cache
            set first packet group's headers with E.Hdr.NMS ← 0 and E.Hdr.NME ← 1
            set last packet group's headers with E.Hdr.NMS ← 1 and E.Hdr.NME ← 0
            all others E.Hdr.NMS ← 1 and E.Hdr.NME ← 1
        else
            /* Send directly from cache */
            TR.DirectCache = TRUE
    end if
```

```
Set Transmission Mask (TR, i)
    /* set the transmission mask to indicate all the packets to be transmitted in */
    /* this packet group */
    if  TR.DirectCache = FALSE
        then
            determine how many elements are in the buffer for the ith packet group
             set Group Transmission Mask (i)
        else
            determine how many elements are in the cache for the ith packet group
            set Group Transmission Mask (i)
    end if


Send Packet (TR)
    /* if nothing {ACK or NACK} is waiting to be retransmitted */
    if  |TR.RTQ| = 0  and TR.State = Transmitting
        then
            /* send directly from cache */
            if  TR.DirectCache = TRUE
                then
                    /* are there more packets to send from the cache */
                    if  TR.CurrentIndex < Cache[TR.CacheIndex].NumPackets
                        /* transmit next packet in the group (if any) */
                        /* otherwise move onto next group */
                        if  TR.CurrentIndex  mod (MaxSegsPerGroup-1)= 0
                            then
                                TR.GSN ← TR.GSN ⊕ 1
                                Set Transmission Mask (TR, TR.GSN)
                        end if
                        clear TR.GrpTransmissionMask bit
                            TR.CurrentIndex mod (MaxSegsPerGroup-1)

                        /* transmit CurrentIndex cache element */
                        Send Packet Directly From Cache (TR, TR.CurrentIndex)
                        TR.CurrentIndex ← TR.CurrentIndex + 1
                    end if
                else
```

```
                /* not directly from cache, are there more packets to send */
                if  |TR.TPGB[GSN]| > 0
                    then
                        /* transmit next packet in the group (if any) */
                        /* find first bit set in this group otherwise move onto next grp*/
                        i ← find first TR.GrpTransmissionMask bit set
                        if  i <0
                            then
                                TR.GSN ← TR.GSN ⊕ 1
                                Set Transmission Mask (TR, TR.GSN)
                        end if
                        clear TR.GrpTransmissionMask bit i
                        TR.P ← TR.TPGB[GSN,i]
                end if
            end if
            /* set the data rate timer—expiration indicates the server is ready */
            /* to send another packet */
            TR.TS4 ← TR.IPG
            TR.P.Hdr.IPG ← TR.IPG
            /* checksum and send the packet */
            checksum TR.P
            send TR.P
        else
            /* set the data rate timer—expiration indicates the client is */
            /* ready to send another packet */
            TR.TS4 ← TR.IPG
            /* remove head of retransmit queue and send it */
            tempP ← head(TR.RTQ)
            tempP.Hdr.IPG ← TR.IPG
            /* checksum and send the packet */
            checksum tempP
            send tempP
    end if

End Transmission (TR)
    TR.State ← Waiting
    /* remaining lifetime */
    TR.TS3 ← MSL + MaxRetries_NACK · 2 MSL

/* transaction timer */
TS1 for transaction record TRs expires:
    TRs.TS1 ← ∞
    TRs.State ← Waiting
    FAILURE
    end if
```

```
/* reception timer */
TS2 for transaction record TRs expires:
    TRs.TS2 ← ∞
    /* expected packet was not received within IPGmax, send NACK */
    tempP.Hdr.SegmentSize← 0
    tempP.Hdr.ServerAddr← TRs.ServerAddr
    tempP.Hdr.ServerPort← TRs.ServerPort
    tempP.Hdr.ClientAddr← TRs.HostAddr
    tempP.Hdr.ClientPort← TRs.HostPort
    tempP.Hdr.TI ← TR.TI
    tempP.Hdr.GSN← TRs.GSN
    i ← find next bit in TRs.DeliveryMask
    tempP.Hdr.GroupMask← 2^i
    tempP.Hdr.NACK ← 1
    tempP.Hdr.ACK ← 0
    add tempP to head of TRs.RTQ
    /* expect NACK within the estimate of RTT */
    TRs.RetransmissionCount[TRs.GSN, i] ← TR.RetransmissionCount[TRs.GSN, i] + 1
    if TRs.RetransmissionCount[TRs.GSN, i] > MaxRetries
        then
            TRs.TS1 ← ∞
            TRs.State ← Waiting
            FAILURE
        else
            TRs.TS2 ← RTT
            /* increment lifetime timer */
            TRs.TS3 ← TRs.TS3 + TRs.RTT
            /* send the NACK */
            Send Packet (TRs)
    end if


/* lifetime timer */
TS3 for transaction record TRs expires:
    TRs.TS3 ← ∞
    if TRs.State = Waiting
        then
            SUCCESS
            release transaction record TRs
        else
            FAILURE
    end if
```

/* data rate timer */

TS_4 for transaction record TR_s expires:

$TR_s.TS_4 \leftarrow \infty$

if *(TR_s.DirectCache=TRUE and TR_s.CurrentIndex < Cache[TR_s.CacheIndex].NumPackets)* **or** $|TR_s.RPBQ[GSN]|>0$ **or** $|TR_s.RTQ|>0$

then

Send Packet (TR_s)

else

End Transmission (TR_s)

end if

***NACK* is received (P)**

$TR \leftarrow TR_s$(P.Hdr.ClientAddr, P.Hdr.ClientPort, P.Hdr.TI)

/* reset lifetime timer */

$TR.TS_3 \leftarrow \infty$

$TR.IPG \leftarrow TR.IPG + IPG_{Adjustment}$

if $TR.IPG > IPG_{max}$

then

FAILURE

end if

if *DirectCache=FALSE*

then

add *TR.RPGB[P.Hdr.GroupMask, P.Hdr.GSN] to TR.RTQ*

else

add *Cache[TR.CacheIndex, x] to TR.RTQ, where x is position in the cache for packet {P.Hdr.GroupMask, P.Hdr.GSN}*

/* if ready to send, send the packet */

if $TR.TS_4 = \infty$

then

Send Packet (*TR*)

end if

```
Send ACK (TR)
    if  TR.AckRequestCount > MaxRetriesAckReq
        then
            /* halt Transmissions and wait until lifetime expires */
            FAILURE
            TR.TS4 = ∞
            TR.State = Waiting
        else
            TR.AckRequestCount ← TR.AckRequestCount + 1
            /* send ACK */
            tempP.Hdr.SegmentSize← 0
            tempP.Hdr.ServerAddr← TR.ServerAddr
            tempP.Hdr.ServerPort← TR.ServerPort
            tempP.Hdr.ClientAddr← TR.HostAddr
            tempP.Hdr.ClientPort← TR.HostPort
            tempP.Hdr.TI ← TR.TI
            tempP.Hdr.GSN← 0
            tempP.Hdr.NACK ← 0
            tempP.Hdr.ACK ← 1
            add P to head of TR.RTQ
            Send Packet (TR)
    end if


ACK is received (P)
    TR ← TRs(P.Hdr.ClientAddr, P.Hdr.ClientPort, P.Hdr.TI)
    if  TR.State=Processing
        then
            Send ACK (TRs)
    end if
```

Client

The transaction record TR_c at the client contains the following components for each transaction:

TI	/* transaction identifier */
RTT	/* estimate of Round Trip Time */
SPT	/* estimate of Server Processing Time */
TC_1	/* transmission timer */
TC_2	/* reception timer */
TC_3	/* lifetime */
TC_4	/* data rate timer */
RetransmissionCount	/* retransmission counter (for each packet)*/
AckRequestCount	/* ACK request counter */
RPGB	/* reception buffer of packet groups */
WQ	/* waiting queue*/
RTQ	/* retransmission queue of packet groups */
State	/* state = {*Transmitting, WaitingForResponseReceiving, WaitingForDuplicates*} */
P	/* a packet */
PacketCount	/* number of packets received so far */
ServerAddr	/* server's address */
ServerPort	/* server's port */
ClientAddr	/* client's address */
ClientPort	/* client 's port */
GrpTransmissionMask	/* bit mask indicating the portions of the group transmitted; */ /* set before entering the transmission queue and cleared */ /* incrementally as the 512-byte packets of the group */ /* are transmitted. */
GSN	/* group sequence number */

Initialization:

TI ← transaction identifier generated from 32-bit clock
IPG ← system default interpacket gap
IPG_{min} ← system default minimum interpacket gap
IPG_{max} ← system default maximum interpacket gap
$IPG_{Adjustment}$ ← system default interpacket gap adjustment
SPT ← default estimate of server processing time
RTT ← default estimate of round trip time
HostAddr ← host's Internet address
MSS ← *512*
MSL ← *120 s*
MaxSegsPerGroup ← *32*
MaxRetries ← *3*
HeaderSize ← *24*
MTU ← *536* (IP), *1536* (Ethernet)
MaxRetriesAckReq ← *3*

Transaction Setup (*port, ServerAddr, ServerPort, msg*):

$TI \leftarrow TI \oplus 1$

Allocate a transaction record TR_c *(ServerAddr, ServerPort, TI)*

$TR_c.ClientPort \leftarrow port$

$TR_c.ClientAddr \leftarrow HostAddr$

$TR_c.ServerAddr \leftarrow ServerAddr$

$TR_c.ServerPort \leftarrow ServerPort$

$TR_c.SPT \leftarrow SPT$

$TR_c.RTT \leftarrow RTT$

$TR_c.IPG \leftarrow IPG$

$TR_c.State \leftarrow Transmitting$

$TR_c.RetransmissionCount \leftarrow 0$

$TR_c.AckRequestCount \leftarrow 0$

$TR_c.TC_1 \leftarrow \infty$

$TR_c.TC_2 \leftarrow \infty$

$TR_c.TC_3 \leftarrow \infty$

$TR_c.TC_4 \leftarrow \infty$

$TR_c.GSN \leftarrow 0$

Send Packet Groups (TR_c, *msg*)

Send Packet Groups (*TR, msg*)

/* initialize default Packet Header */

$TR.P.Hdr.SegmentSize \leftarrow 0$

$TR.P.Hdr.ServerAddr \leftarrow TR.ServerAddr$

$TR.P.Hdr.ServerPort \leftarrow TR.ServerPort$

$TR.P.Hdr.ClientAddr \leftarrow TR.HostAddr$

$TR.P.Hdr.ClientPort \leftarrow TR.ClientPort$

$TR.P.Hdr.TI \leftarrow TR.TI$

$TR.P.Hdr.GSN \leftarrow 0$

$TR.P.Hdr.Data \leftarrow 0$

$TR.P.Hdr.ACK \leftarrow 0$

$TR.P.Hdr.NACK \leftarrow 0$

$TR.P.Hdr.NMS \leftarrow 1$

$TR.P.Hdr.NME \leftarrow 1$

Queue packet groups *(TR, msg)*

$TR.GSN \leftarrow 0$

Set Transmission Mask *(TR, TR.GSN)*

Send Packet *(TR, TR.GSN)*

Queue Packet Groups (*TR, msg*)

create a queue of *m* packet groups *TR.TPGB[m,j]* and their *j* elements (*E*) of packets
each packet may contain more than one cache segment
one packet may contain at most one entire packet group [all *MaxSegsPerGroup* segments]
/* update delivery mask for segment *i = 0..(MaxSegsPerGroup-1)* in the group added */
TR.TPGB[x,y].Hdr.GroupMask ← *TR.TPGB[x,y].Hdr.GroupMask* + 2^i
TR.TPGB[x,y].Hdr.SegmentSize ← *TR.TPGB[x,y].Hdr.SegmentSize* + *segmentSize*
TR.Data ← *TR.Data* + *msg(offset,segmentSize)*
if last packet in last packet group, *P.Hdr.GroupMask* ← *1*
set first packet group's headers with *TR.TPGB[x,y]..Hdr.NMS* ← *0* and
TR.TPGB[x,y].Hdr.NME ← *1*
set last packet group's headers with *TR.TPGB[x,y].Hdr.NMS* ← *1* and
TR.TPGB[x,y].Hdr.NME ← *0*
all others *TR.TPGB[x,y].Hdr.NMS* ← *1* and *TR.TPGB[x,y].Hdr.NME* ← *1*

Set Transmission Mask (*TR, i*)

determine how many elements are in the buffer for the *i*th packet group
set Group Transmission Mask (*i*)

Send Packet (*TR*)

```
/* if nothing waiting to be retransmitted */
if |TR.RTQ| = 0
    then
        if |TR.TPGB[GSN]| > 0
            then
                /* transmit next packet in the group (if any) */
                /* find first bit set in this group otherwise move onto next grp*/
                i ← find first TR.GrpTransmissionMask bit set
                if i <0
                    then
                        TR.GSN ← TR.GSN ⊕ 1
                        Set Transmission Mask (TR, TR.GSN)
                end if
                clear TR.GrpTransmissionMask bit i
                /* set the data rate timer—expiration indicates the client is ready */
                /* to send another packet */
                TR.TC4 ← TR.IPG
                TR.P.Hdr.IPG ← TR.IPG
                /* checksum and send the packet */
                checksum TR.P
                send TR.P
        end if
```

```
else
    /* set the data rate timer—expiration indicates the client is */
    /* ready to send another packet */
    TR.TC4 ← TR.IPG
    /* remove head of retransmit queue and send it */
    tempP ← head(TR.RTQ)
    tempP.Hdr.IPG ← TR.IPG
    send tempP
end if
```

Receive Response Packet (*P*)

```
/* does current source (port and address) and P.Hdr.TI already have a TR */
if not TRc (P.Hdr.ServerAddr,P.Hdr.ServerPort,P.Hdr.TI) exists
    then
        discard P
    else
        TR ← TRc (srcAddr,srctPort,P.TI)
        if TR.State = WaitingForResponse
            then
                TR.PacketCount← 0
                TR.State = Receiving
        end if

        if TR.State = Receiving
            then
                /* increment count of packets received, update upper bound on */
                /* server's lifetime, reset reception timer, and either place in buffer in the */
                /* correct spot or place in the waiting queue */
                TR.PacketCount← TR.PacketCount + 1
                TR.TC3 ← (P.Hdr.MessageLength-TR.PacketCount)·IPGmax + MSL
                /* reset reception timer */
                TR.TC2 ← IPGmax
                /* if packet belongs to current group */
                if P.GSN = TR.GSN
                    then
                        /* add to reception buffer and update delivery mask */
                        TR.RPBQ[i,P.Hdr.GroupMask] = P
                        TR.DeliveryMask ← TR.DeliveryMask + P.Hdr.GroupMask
                        if TR.DeliveryMask = 1
                            then
                                TR.GSN ← TR.GSN ⊕ 1
                                move any TR.WQ packet's with TR.GSN to
                                    TR.RPBQ[GSN]
                        end if
```

```
            /* otherwise add to Waiting Queue */
            else
                add P to TR.WQ
            end if
        /* if the last packet, all done*/
        if  not P.Hdr.NME and TR.DeliveryMask = 1
            /* clear reception timer, set transaction timer */
            TR.TC2 ←∞
            TR.TC3 ←∞
            TR.State ← WaitingForDuplicates
        end if
    end if
```

Request For *ACK* (*TR*)

```
    P.Hdr.SegmentSize← 0
    P.Hdr.ServerAddr← TR.ServerAddr
    P.Hdr.ServerPort← TR.ServerPort
    P.Hdr.ClientAddr← TR.ClientAddr
    P.Hdr.ClientPort← TR.ClientPort
    P.Hdr.TI ← TR.TI
    P.Hdr.GSN← 0
    P.Hdr.NACK ← 0
    P.Hdr.ACK ← 1
    add P to head of TR.RTQ
    Send Packet (TR)
```

***ACK* is received (*TR*)**

```
    TR.AckRequestCount ← 0
    TR.TC1 ← RTT + SPT
```

***NACK* is received (P)**

```
    TR ← TR(P.Hdr.ServerAddr, P.Hdr.ServerPort, P.Hdr.TI)
    /* reset lifetime timer */
    TR.TC3 ← ∞
    TR.IPG ← TR.IPG + IPGAdjustment
    if  TR.IPG > IPGmax
        then
            FAILURE
    end if
    add TR.RPGB[P.Hdr.GroupMask, P.Hdr.SN] to TR.RTQ
    if  TR.TC4 =∞
        then
            Send Packet (TR)
    end if
```

End Transmission (*TR*)

$TR.State \leftarrow WaitingForResponse$

$TR.AckRequestCount \leftarrow 0$

$TR.TC_1 \leftarrow RTT + SPT$

$TR.TC_3 \leftarrow MSL + max(MSL + SPT, MaxRetries_{NACK} \cdot 2\ MSL) + Retries_{ACKreq} \cdot 2\ MSL$

/* transaction timer */

TC_1 for transaction record TR_c expires:

$TR_c.TC_1 \leftarrow \infty$

$TR_c.AckRequestCount \leftarrow TR_c.AckRequestCount + 1$

if $TR_c.AckRequestCount > MaxRetriesAckReq$

then

FAILURE

else

$TR_c.TC_1 \leftarrow TR_c.RTT$

/* increment lifetime timer */

$TR_c.TC_3 \leftarrow TR_c.TC_3 + 2\ MSL$

Request For ACK (TR_c)

end if

/* reception timer */

TC_2 for transaction record TR_c expires:

$TR_c.TC_2 \leftarrow \infty$

/* send NACK */

/* expect NACK within the estimate of RTT */

$TR_c.RetransmissionCount[GSN,i] \leftarrow TR_c.RetransmissionCount[GSN,i] + 1$

if $TR_c.RetransmissionCount[GSN,i] > MaxRetries$

then

$TR_c.TC_1 \leftarrow \infty$

$TR_c.State \leftarrow Waiting$

FAILURE

else

$TR_c.TC_2 \leftarrow TR_c.RTT$

/* increment lifetime timer */

$TR_c.TC_3 \leftarrow TR_c.TC_3 + TR_c.RTT$

/* send the NACK */

Send Packet (TR_c)

end if

```
/* lifetime timer */
TC3 for transaction record TRc expires:
    TRc.TC3 ← ∞
    if   TRc.State = WaitingForDuplicates
         then   SUCCESS
                release transaction record TRc
         else   FAILURE
    end if


/* data rate timer */
TC4 for transaction record TRc expires:
    TRc.TC4 ← ∞
    if   |TRc.RPBQ[GSN]|>0 or |TRc.RTQ|>0
         then
              Send Packet (TRc)
         else
              End Transmission (TRc)
    end if
```

HTTP Server

Process Request (***requestMsg, responseMsg***)

```
    Determine Requested Object(s)
    /* Check if objects(s) are present in the cache */
    index ← ODTP Server Find Cache Index (objects)
    if  index<0
        responseMsg ← Process Object (objects)
        return -1
    else
        latest ← ODTP Server Is Latest Version (objects, objects.timestamp)
        if  latest=FALSE
            then
                responseMsg ← Process Object (objects)
                return -1
            else
                return index
        end if
    end if
```

Analysis/Conclusion

Through the analysis above, this research established the deficiencies of the current HTTP/TCP interaction, demonstrated how existing solutions do not fully address the needs of the World-Wide Web, and established the three objectives for an efficient transport protocol for the WWW. These objectives for a new transport protocol to address the performance problems of HTTP/TCP were:

1. A transaction rather than stream-based protocol -- this implies a low connection management overhead on connection establishment and a minimum transaction latency of one RTT for random objects regardless of prior readership
2. Effective connection avoidance for high-speed transmission rates and short connection that will allow the protocol to scale with improvements in network bandwidth.
3. Minimization of server load and server processing time (SPT)

The final contribution of this research was the specification of a new transport protocol that addresses these objectives. The new ODTP protocol provides efficient connection-oriented, transport services for message transactions between a client and server. ODTP utilizes the naming (port and address) from TCP, timer-based connection management from Delta-t and Fletcher, rate-based flow control from VMTP, and an unique packet response cache that requires a modified HTTP server. A proof of correctness of the protocol's connection management and a formal description of the protocol and the HTTP modifications are provided.

ODTP specifies the following features to address all the objectives above:

1. ODTP is a transaction-oriented transport protocol modeled after VMTP—but ODTP uses timers exclusively to handle duplicate data detection. Hence, this reduced connection management overhead leads to the minimum transaction latency for ODTP of one RTT plus SPT for random object access regardless of prior readership.

2. ODTP like VMTP incorporates rate-based flow control for effective congestion avoidance for high-speed transmission rates and short connections.
3. Through a response cache, ODTP places minimal responsibility on the HTTP and ODTP servers for handling the requests. If objects are added to the cache when they are published online, all responses can be served from the cache. The cache also enables ODTP servers to forgo a transmission buffer and send directly from the cache and also access the cache directly for retransmissions.

Stage	Client	Server	Comments
1. one RTT	REQ		ODTP
		RESP	

ODTP provides reliable sequenced delivery of both the request and the response by using sequencing, transaction identifiers, checksums, and negative acknowledgment and timeout and retransmission of missing packets. Overall, these enhancements also lessen the burden on the HTTP server and increase server scalability by reducing the per-connection information that the server must maintain (which provide no performance benefits). Thus, ODTP is one specification for a simple transaction protocol for HTTP for delivering documents efficiently over both networks with high bandwidth and/or long round-trip delays and conventional Local and Wide Area Networks.

Bibliography

Abrams, Marc, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. "Caching Proxies: Limitations and Potentials." *Proceedings of the Fourth International World-Wide Web Conference* (December 1995).

Adie, Chris. *Network Access to Multimedia Information.* Second edition. Amsterdam: RARE Project, 1993.

Berners-Lee, Tim. "The HTTP Protocol as Implemented in W3." (1992).

Berners-Lee, Tim, R. Cailliau, J. Groff, and B. Pollermann. "World-Wide Web: The information universe," *Electronic Networking: Research, Applications, and Policy* 1.2 (1992): 52-58.

Berners-Lee, Tim, R. Cailliau, A. Luotonen, H. Nielsen, and A. Secret. "The World-Wide Web." *Communications of the ACM* 37.8 (1994): 76-82.

Berners-Lee, Tim, Roy T. Fielding, and Henrik Frystyk Nielsen. "Hypertext Transfer Protocol—HTTP/1.0." Fourth Edition. (1995).

Berners-Lee, Tim. "Propagation, Replication, and Caching." *W3C Proposal* (March 15, 1995).

Berners-Lee, Tim, Roy T. Fielding, and Henrik Frystyk Nielsen. "Hypertext Transfer Protocol—HTTP/1.1." First edition. (1996).

Biersack, Ernst W. and David C. Feldmeier. "A Timer-Based Connection Management Protocol with Synchronized Clocks and Its Verification." (July 7, 1992).

Braden, Richard. "Requirements for Internet Hosts—Communication Layers." RFC 1122. (1989).

Braden, Richard. "T/TCP—TCP Extensions for Transactions Functional Description." RFC 1644. (July 1994).

Brakmo, Lawrence S., Sean W. O'Malley, and Larry L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance." *Proceedings of the SIGCOMM '94 Symposium* (August 1994): 24-35.

Cheriton, David. "VMTP: Versatile Message Transaction Protocol." RFC-1045. (February 1988).

Clark, David, Mark Lambert, and Lixia Zhang. "NETBLT: A Bulk Data Transfer Protocol." RFC-988. (March 1989).

Clark, Russell J. and Mostafa H. Ammar. "Providing Scalable Web Services Using Multicast Communication." *Proceedings of the IEEE Workshop on Services in Distributed and Networked Environment* (June 1995).

Crovella, Mark E. and Azer Bestavros. "Explaining World Wide Web Traffic Self-Similarity." *Technical Report TR-95-015*, Boston University Computer Science Department (October 12, 1995).

Dabbous, Walid S. "On High Speed Transport Protocols." *Protocols for High-Speed Networks.* Eds. H. Rudin and R. Williamson. North Holland: Elsevier Science Publishers B.V., 1989.

Doeringer, W.A., H. D. Dykeman, M. Kaiserswerth, B. W. Meister, H. Ruden, and R. Williamson. "A Survey of Light-Weight Protocols For High-Speed Networks." *High Performance Networks: Technology and Protocols* Ed. Ahmed N. Tantaway. Boston: Kluwer Academic Publishers, 1994.

Fall, K. and S. Floyd. "Comparisons of Tahoe, Reno, and Sack TCP." (December 1995).

Feldmeier, David C. "An Overview of the TP++ Transport Protocol." *High Performance Networks: Frontiers and Experience*. Ed. Ahmed N. Tantawy. Boston: Kluwer Academic Publishers, 1994.

Fletcher, John G. and Richard W. Watson. "Mechanism for a Reliable Timer-Based Protocol." *Computer Networks* 2 (1978): 271-290.

Floyd, S. "Issues of TCP with SACK." (January 1996).

Fox, Richard. "TCP Big Window and NAK Options." RFC 1106. (June 1989).

Glassman, Steve. "A Caching Relay for the World-Wide Web." *Proceedings of the First International Conference on the WWW* (May 1994).

Heywood, Drew, Janos (John) Jerney, Jon Johnson, *et al. Connectivity: Local Area Networks*. Carmel: New Riders Publishing, 1992.

Jacobson, Van. "Congestion Avoidance and Control." *Proceedings of SIGCOMM '88* (August 1988).

Jacobson, Van. "Modified TCP Congestion Avoidance Algorithm." (April 1990).

Jacobson, Van. "Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno." *Proceedings of the Eighteenth Internet Engineering Task Force* (September 1990).

Jacobson, Van, and R. Braden. "TCP Extensions for Long-Delay Paths." RFC 1072. (October 1988).

Jacobson, Van, R. Braden, and D. Borman. "TCP Extensions for High Performance." RFC 1323. (May 1992).

Jain, R. "A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks." *ACM Computer Communications Review* 19.5 (October 1989): 56-71.

Luotonen, A. and K. Altis. "World-Wide Web Proxies." *Proceedings of the First International Conference on the WWW* (May 1994).

Mathias, Matt, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. "TCP Selective Acknowledgment Options." First Edition Draft. (January 24, 1996).

Mishra, Partho P., Dheeraj Sanghi, and Satish K. Tripathi. "TCP Flow Control in Lossy Networks: Analysis and Enhancement." *Computer Networks: Architecture and Applications* Eds. S.V. Raghavan, G. v. Bochmann, and G. Pujolle. North Holland: Elsevier Science Publishers B.V., 1993.

McKenzie, Alex. "A Problem with the TCP Big Window Option." RFC 1110. (August 1989).

Mill, D.L. "Internet Time Synchronization: The Network Time Protocol." *IEEE Transactions on Communication* 39.10 (October 1991): 1482:1493.

Nagle, J. "Congestion Control in IP/TCP Internetworks." RFC 896. (January 1984).

Padmanabhan, Venkata N. and Jeffrey C. Mogul. "Improving HTTP Latency." *Second International WWW Conference* (July 1994).

Pam, Andrew. "Where World Wide Web Went Wrong." *Asian Pacific World Wide Web Regional Conference* (September 1995).

Pink, Stephen. "TCP/IP on Gigabit Networks" *High Performance Networks: Frontiers and Experience*. Ed. Ahmed N. Tantawy. Boston: Kluwer Academic Publishers, 1994.

Pitkow, James E. and Margaret M. Recker. "A Simple Yet Robust Caching Algorithm Based on Dynamic Access Patterns." *Proceedings of the Second International WWW Conference* (1994).

Postel, J. (ed.). "Internet Protocol—DARPA Internet Program Protocol Specification." RFC 791. (September 1981).

Postel, J. (ed.). "Transmission Control Protocol—DARPA Internet Program Protocol Specification." RFC 793. (September 1981).

Sanghi, Dheeraj and Ashok K. Agrawala. "DTP: An Efficient Transport Protocol." Computer Networks: Architectures and Applications. Eds. S.V. Raghavan, G. v. Bochmann, and G. Pujolle. North Holland: Elsevier Science Publishers B.V., 1993.

Shankar, A. Udaya and David Lee. "Minimum-Latency Transport Protocols with Modulo-N Incarnation Numbers." *Technical Report UMIACS-TR-93-24.1*, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland (December 15, 1994).

Smith, N. "What can Archives Offer the World-Wide Web?" *Proceedings of the First International World-Wide Web Conference* (March 1994).

Spasojevic, Mirjana, Mic Bowman, and Alfred Spector. "Using a Wide-Area File System Within the World-Wide Web." *Second International WWW Conference* (July 1994).

Spero, Simon E. "Analysis of HTTP Performance Problems" (July 1994).

Stevens, W. Richard. "TCP/IP Illustrated, Volume 1." Reading: Addison-Wesley Publishing Company, Incorporated, 1994.

Tanenbaum, Andrew S. *Computer Networks*. Englewood Cliffs: P T R Prentice Hall, 1989.

Tawbi, Wassim, Sylvie Dupuy, and Eric Horlait. "High Speed Protocols: State of the Art in Multimedia Applications." *Information Network and Communications IV.* Eds. Martti Tienari and Dipak Khakhar (March 1992).

Wang, Z. and J. Crowcroft. "A New Congestion Control Scheme: Slow Start and Search (Tri-S)." *ACM Computer Communications Review* 22.2 (April 1992): 9-16.

Watson, Richard W. "The Delta-T Transport Protocol: Features and Experience." *Protocols for High-Speed Networks*. Eds. H. Rudin and R. Williamson. North Holland: Elsevier Science Publishers B.V., 1989.

Zhang, Lixia, Scott Shenker, and David D. Clark. "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic." *Proceedings of SIGCOMM '91 Conference, Computer Communications Review* 21:4 (September 1991): 133-147.

Vita

Born October 6, 1972 in Allentown, PA to Janice R. Moore and Andrew H. R. Moore

Graduate of Parkland High School (Orefield, PA) in June 1990

Graduate of Ursinus College (Collegeville, PA) in May 1994 with a Bachelor of Science Degree in Mathematics/Computer Science and Economics

- Valedictorian
- *Summa cum laude*
- *Phi Beta Kappa*
- Honors in Economics, "Optimal Production Portfolio Theory"
- Best paper award (1992) for "The United States Income Tax: The Effects of the 1986 Tax Reform Act on Lawyers" at Regional Undergraduate Economics Conference

Experience

RPR Pharmaceuticals, Collegeville, PA

- C Programmer (May 1993-May 1994)

Ursinus College, Collegeville, PA

- Administrative Computing User Support Assistant (Sept 1990-Sept 1992)
- Network System Assistant (Sept 1992-Aug 1994)

Lehigh University, Bethlehem, PA

- Research Assistant for Network Management (Sept 1994-Dec 1994)
- Research Assistant for Advanced Manufacturing Software (Jan 1995-present)

END OF TITLE