Chapter 3

Methodology

In this chapter, we describe our research methodology. Given the wide diversity of networks and dynamically-varying traffic patterns in the Internet, it is difficult to conduct networking research in a limited environment and yet claim with a high degree of confidence that the results are generally applicable. Therefore, as we will discuss next, we use a broad-based methodology to maximize the confidence in our research results.

We begin with an overview of our methodology. In Section 3.1, we describe our experimental testbed containing a heterogeneous collection of access networks. This provides use In Section 3.2, we discuss the traffic traces we gathered and analyzed. Section 3.3 discusses the network simulator we used to evaluate our algorithms. Finally, we discuss our implementation platform in Section 3.4.

Since this thesis addresses the challenges of Web data transport, it is natural that the first step should be to identify these challenges. There are two ways in which we do this. The first was to characterize performance by conducting live experiments across the Internet and over a variety of access networks in our experimental testbed (Section 3.1). The second was to collect and analyze traffic traces at a busy Web server (Section 3.2).

Once we had identified performance problems, we designed new algorithms to overcome them. During this phase, we conducted simulation experiments to explore the design space (Section 3.3). We followed this up with an implementation of the algorithms on an actual network platform Section 3.4). This enabled us to identify, and if necessary re-design, any features of the algorithms that made implementation difficult. Finally, we evaluated the algorithms via extensive experiments in the simulator as well as our network testbed (Section 3.5).

We used a number of software tools, both existing ones and new ones that we developed, to facilitate our analysis and evaluation. These are described in Appendix A.

3.1 Experimental Network Testbed

We used a heterogeneous network testbed for our experiments. The goal of these experiments was to analyze the dynamics and interaction of the TCP and the HTTP protocols in detail, and to identify performance problems that arise. Our testbed enabled us to conduct controlled experiments with real implementations of these protocols. The specific network characteristics of interest in our investigation (and the networks with those characteristics in our testbed) include:

1. Large delay and/or large delay-bandwidth product (DirecPC satellite network).

2. Bandwidth asymmetry (Hybrid wireless cable modem network).

3. Low bandwidth (dialup phone line, Ricochet packet radio network).

The networks in our testbed are examples of emerging access network technologies outlined in Section 1.6. We briefly describe the key characteristics and the architecture of each network.

3.1.1 DirecPC Satellite Network

DirecPC direct broadcast satellite (DBS) network [27] is an example of a satellite-based network dedicated to providing high-speed data access to residential and commercial users alike. The primary advantage of a satellite-based network is its geographical reach and the ease with which connectivity can be provided to a new location once the satellite system is already in place.

Figure 3.1 depicts the architecture of the DirecPC DBS system, which provides *asymmetric* network connectivity to users. The system is centered around a geostationary satellite that receives

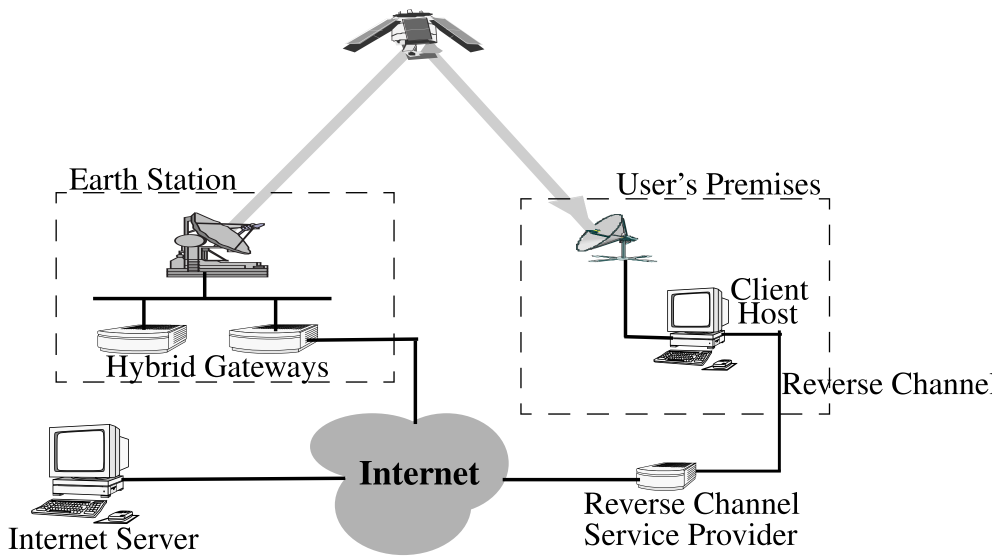

Figure 3.1 The architecture of the DirecPC system.

data from an earth station and broadcasts it *directly* to the users' premises. The bandwidth of this downstream channel is about 12 Mbps and it is shared among all users in a large geographical region via packet-level multiplexing. The advertised bandwidth per client is 400 Kbps, but the a client's bandwidth share could exceed this or fall short of this depending on the total number of clients that are active.

Connectivity in the reverse (upstream) direction is provided by another means, such as a dialup phone line or a terrestrial wireless network. So the data rate of the upstream link tends to be much lower, typically 15-30 Kbps, resulting in a high degree of bandwidth asymmetry.

The *hybrid gateway*, located at the earth station, mediates between the client hosts and other Internet hosts. It participates in routing data packets to and from each client host in an asymmetric manner [83]. A client host tunnel outgoing packets over its upstream link to the hybrid gateway, which then forwards them on to the actual destination host. Incoming packets specify the client host's address in the DirecPC domain as the destination, so get routed via the satellite link automatically.

The geostationary satellite link has a large delay of approximately 240 ms. Any communication between a client host and a server host in the Internet involves additional delays due to the upstream link and the Internet path leading to the server. So in practice the round-trip time for a TCP connection via the DirecPC network tends to be in the 300-400 ms range. This large delay

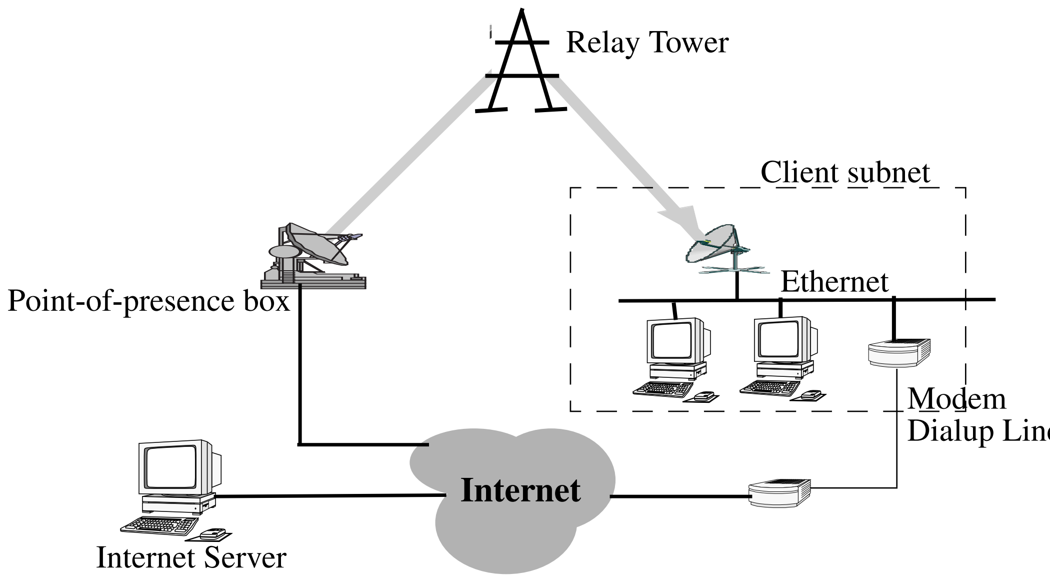

Figure 3.2 The architecture of the wireless cable modem network.

coupled with a relatively high bandwidth results in a large bandwidth-delay product, i.e., a large capacity "data pipe". Web transfers, which tend to be short in length, are unable to utilize this capacity effectively because of the slow start procedure. In Chapter 8, we present our *TCP fast start* technique to improve utilization, especially in networks such as this, by avoiding slow start when possible.

3.1.2 Hybrid Wireless Cable Modem Network

The "wireless cable" modem network from Hybrid, Inc. [47] is a high-speed access network designed to provide connectivity across a metropolitan area. Its architecture, depicted in Figure 3.2, is quite similar to that of the DirecPC network except on a smaller scale. A terrestrial transmission tower takes the place of the satellite and a point-of-presence (PoP) box that of the hybrid gateway.

Like the DirecPC network, this network is also asymmetric. It uses a multi-point multi--channel distribution (MMDS) link operating in the 2.4 GHz band to provide a 10 Mbps downstream broadcast channel. Upstream connectivity is provided via another means such as a dialup phone line. The delay of the wireless downlink is less than 5 ms, so the delay of the upstream link and that of the Internet tend to dominate.

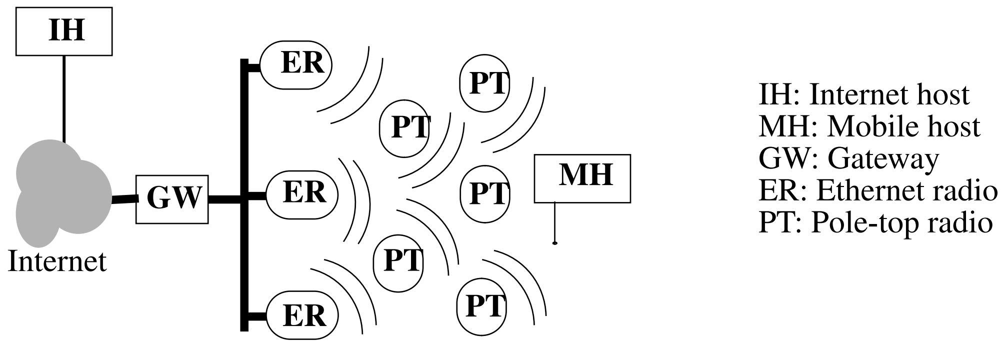

Figure 3.3 Topology of the Ricochet packet radio network.

Although the downlink is wireless, it is almost error-free owing to the fixed locations of the transmitter and the receiver while active[1]. So the link being wireless has little impact on the performance of TCP. The more important factor impacting performance is the high degree of asymmetry. This makes the upstream link prone to congestion which can degrade downstream throughput even when there is no congestion in that direction. We discuss this problem and develop solutions for it in Chapter 9. Given that it is asymmetry and not the wireless connectivity that impacts performance, we believe that our results extend to the (far more common) wireline one-way cable modem-based networks as well.

3.1.3 Ricochet Packet Radio Network

Another network in our testbed is the Ricochet packet radio network from Metricom, Inc. [71]. This is a wireless data network that provides connectivity to mobile users, much like the Cellular Digital Packet Data (CDPD) [17] and Global System for Mobile Communications (GSM) [42] networks.

The Ricochet network model is shown in Figure 3.3. It consists of *portable* radios that are carried around by users and fixed-location *pole-top* radios. A subset of the pole-top radios, called *ethernet* radios, have a wireline network connection that gateways into the Internet. Communication between a mobile host (connected to a portable radio) and a host on the Internet usually requires traversing multiple wireless hops (in addition to the wireline path through the Internet).

---

1. It is fairly easy to relocate the receiver, but doing so would lead to temporary network disconnection.

The key characteristics of the Ricochet network that is of interest to us are its low data rate and its large and variable latency. Although raw data rate of the channel is 100 Kbps, the effective data rate is only around 30-40 Kbps because of multiple hops in the wireless network and other sources of overhead. The one-way delay through the wireless network is of the order of 150-200 ms. But at times this can be a factor of 2-4 larger because of media access contention.

The low data rate of the network makes it prone to congestion. In Chapter 6, we discuss how our TCP session technique can alleviate congestion and improve throughput. The media access contention is aggravated when the flow of traffic is bidirectional, for example, TCP data packets flowing in one direction and acks in the other. As we showed in joint work with Balakrishnan [5], the techniques that help alleviate the effects of bandwidth asymmetry are also effective in this case. We discuss this briefly in Chapter 9. A more detailed investigation is the subject of a contemporaneous thesis by Balakrishnan [4].

3.2 Web Server Traffic Trace Analysis

In addition to analyzing performance via experiments in our testbed, we analyzed a large packet-level traffic trace. The traces were from the official Web server for the Summer 1996 Olympic Games [50]. This server was run by IBM. We analyzed:

1. The effectiveness of TCP loss recovery in the face of short and bursty Web connections.
2. The impact of multiple concurrent connections between the server and a client on TCP congestion control.

We describe the trace collection and post-processing steps here, and present details of the analysis in Section 4.3.

3.2.1 The IBM 1996 Olympics Web Server

The Olympics Web server experienced very high levels of traffic load and connection rates, and thus made for an ideal case study. To cope with the high level of load, the Web server was mirrored at four locations around the world: Southbury (CT, USA), Cornell (NY, USA), Keio (Japan), and Karlsruhe (Germany). In spite of the mirroring, the primary server site in Southbury experienced nearly 17 million Web hits on the most busy day.

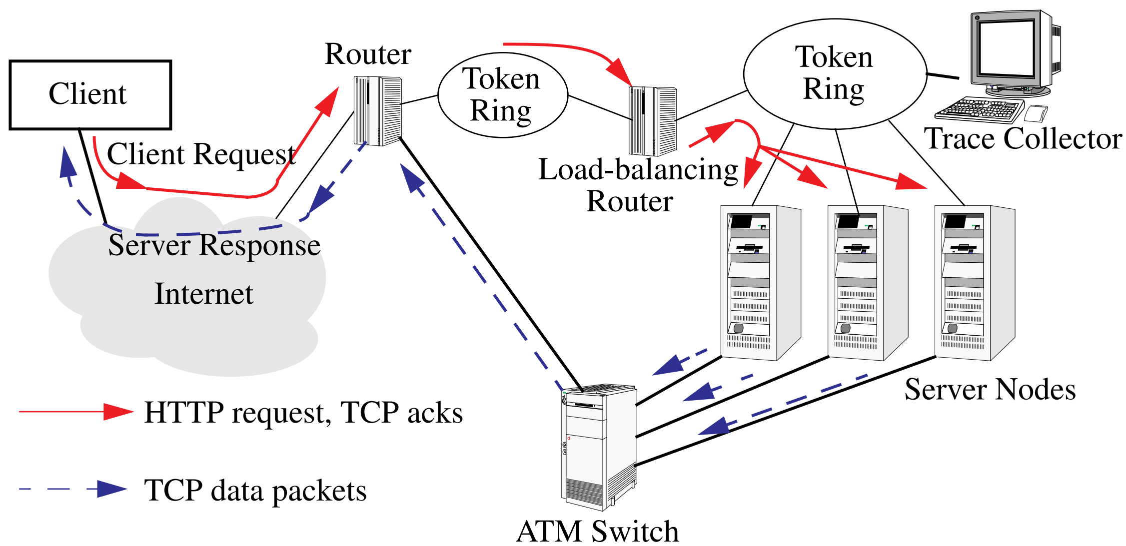

Figure 3.4 The Web server and monitoring setup at the Southbury, CT site.

The architecture and the network topology of the Web server at the primary site in Southbury are shown in Figure 3.4. During the Olympics, the Web site was connected via a T3 link (45 Mbps) to four U.S. Network Access Points (NAPs), located in Chicago (Bellcore and Ameritech), the San Francisco Bay Area (Bellcore and Pacific Bell), New York (Sprint), and Washington, D.C. (MFS Datanet). Requests for data from a client arrived at the Web site after being routed through the appropriate Internet NAP. Each request was subsequently redirected to a load-balancing connection router [48], that distributed them across several server nodes. These nodes then retrieved the appropriate Web objects and transmitted them across an internal ATM network and through the Internet to the clients. More details about the site structure and software are available from [49].

3.2.2 Data Collection Methodology

The primary Web server site was instrumented by Srinivasan Seshan and Mark Stemm to obtain packet-level traffic traces. All the traffic coming into the site was traced by running the *tcpdump* tool on a machine placed on the token ring connecting the load-balancing router to the server nodes (Figure 3.4). This machine was an IBM 150Mhz Pentium Pro PC running BSD/OS 2.1 from BSDI [16]. The first 350 bytes of every packet destined to TCP port 80 (the default HTTP port) were captured in the trace. This was then compressed on-the-fly using the *gzip* utility and periodically dumped to tape. Although the tracing system did not capture traffic (such as streaming audio) on ports other than the default HTTP port, during peak periods it still collected approximately 1 GB of compressed packet headers per hour.

Due to the location of the trace collection machine and the asymmetric path taken by the request and response streams, only packets coming from the clients *into* the Web server complex could be captured. These included the TCP SYN packet from clients at the time of connection setup, the HTTP request packets, and all TCP acknowledgments from clients in response to data sent by the server. Although none of the packets sent by the server to clients were traced, we were able to infer the approximate behavior of the outgoing packets since much of a TCP sender's activity is triggered by incoming ack packets (or the lack of such packets). To facilitate validation of such a reconstruction, we instrumented the TCP stack on a subset of the server nodes to trace the congestion window size, smoothed round-trip time estimate, and number of duplicate acknowledgments whenever a retransmission happened. We discuss the reconstruction procedure in some detail next.

3.2.3 Reconstructing the TCP Packet Trace

Reconstructing the entire TCP packet trace from just a trace of the incoming packets requires a way to faithfully reproduce the state of the sender for each TCP connection. For this purpose, we wrote a *TCP emulation engine* that took the trace of acknowledgments and other incoming packets as input and approximated the evolution of the sender's state variables (such as the congestion window size, the round trip time estimate, maximum packet transmission sequence number, etc.). As we explain below, this is possible to do because changes in sender-side state variables are triggered either by the reception of an ack or the expiration of a timer.

In essence, the engine uses the ack trace to emulate a subset of the functions that a TCP sender performs while an actual connection is in progress. Armed with the knowledge of the algorithms that the sender employs, it is possible to reconstruct the sender's behavior with complete accuracy if the evolution of the sender's state were completely driven by the arrival of acks[2] (as is usually the case if a connection experiences no losses). However, in practice packet losses do happen, sometimes causing the sender to time out. But it is possible to use the ack trace to infer the occurrence of such events as well. Doing so with our trace involved two steps. First, during times when there

2. This also assumes that the server always has data ready to send.

was no packet loss, the ack trace together with the reconstructed data packet trace was used to obtain an estimate of the round trip time and its variance. These were then used to compute the sender's retransmission timeout value (RT0). Second, a retransmission timeout event was flagged whenever more than RTO time had elapsed since the (inferred) transmission of a data packet but no ack had been received for it yet.

We validated the TCP emulation engine by comparing the times at which it predicted a retransmission event with the actual times at which retransmissions occurred, as reported by the server nodes with instrumented network stacks. This comparison revealed that the engine predicts the number of coarse timeouts, fast retransmissions, and slow-start retransmissions to within 0.75% of that recorded by the instrumented network stack. Consequently, the congestion window size computed by the engine is also highly accurate. The only other potential source of error is underlying loss rate in the packet capture process, but this was only about 0.5%.

In summary, our TCP emulation engine reconstructs the evolution of the TCP sender state for each connection between the Web server and clients, with a high degree of accuracy. This facilitates the analysis we present in Section 4.3.

3.3 Simulation Study

Our experimental and trace-based analyses helped us identify performance problems that afflict the TCP and HTTP protocols. In the next phase of our work, we designed new algorithms to address these problems. The algorithms include:

- Persistent-connection HTTP (P-HTTP)
- TCP session
- TCP fast start
- TCP for asymmetric networks

To aid in exploring the design space, and the process of tuning and evaluating new algorithms, we used the *ns* network simulator [81]. This is a packet-level discrete-event simulator that enables detailed simulation of a variety of network protocols using complex network topologies. The simulator comprises two components — a core written in the C++ language and a set of scripts written

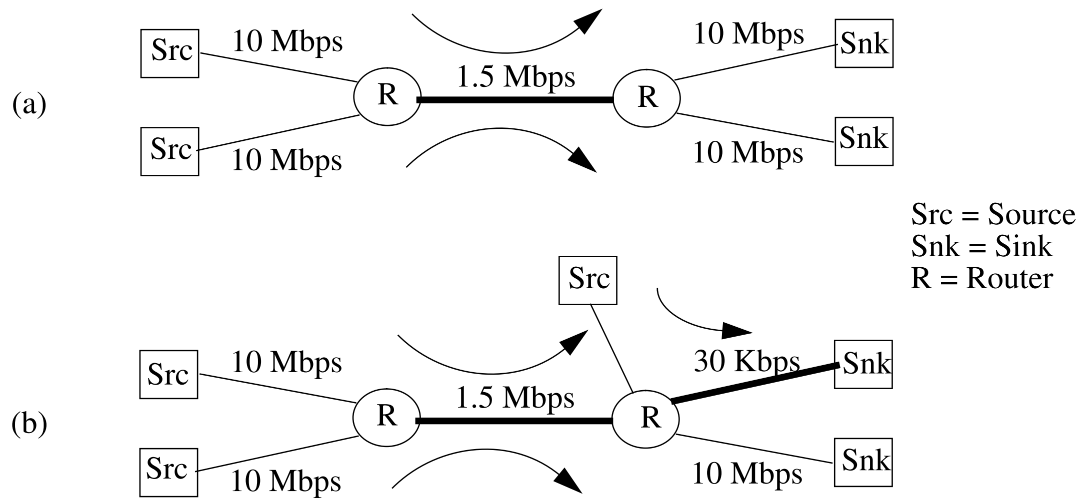

Figure 3.5 Example simulation topologies with (a) a single bottleneck link, and (b) two bottleneck links (shown as thick lines).

in Object Tcl (OTcl) [106]. The C++ core includes modules for a variety of protocols, including several flavors of TCP (Tahoe, Reno, NewReno, etc.). OTcl scripts are used to compose these modules and create simulation topologies.

We extended the C++ core with modules implementing the new algorithms we developed. These include TCP session, TCP fast start and asymmetric TCP. We also implemented link scheduling and buffer management disciplines for use in conjunction with these end-to-end algorithms.

We built a simple Web client and server in OTcl. These use the TCP modules in the C++ core to establish connections for the requests and responses. The client and server can be configured to emulate the HTTP/1.0 protocol, either with sequential or with concurrent connections, or P-HTTP.

The workload we used for simulation experiments was a mix of bursty Web-like traffic and bulk transfers. We used a variety of network topologies. Some of the topologies were patterned after the DirecPC satellite network and the Hybrid wireless cable modem network in our testbed. In addition, we uses simple network topologies with one or more bottleneck links, as illustrated in Figure 3.5. These topologies serve as useful abstractions of complex Internet paths since they capture the essential determinants of performance — the bandwidth and buffer size of the bottleneck link(s), and the end-to-end round trip time.

3.4 Implementation

After refining our new algorithms via simulation, we implemented them in an actual network protocol stack. Our object in doing so was two-fold:

1. To determine the changes, if any, needed to the algorithms to make them suitable for a real implementation.
2. To evaluate the algorithms in our network testbed and identify shortcomings, if any.

Many of our algorithms, including TCP session, TCP fast start and asymmetric TCP, involve changes to the TCP/IP protocol stack. Since the BSD protocol stack is well-documented and has influenced several other protocol stack implementations, we chose BSD/OS 3.0 [16] running on Pentium-based PCs as our software/hardware platform.

Some of our algorithms, for instance P-HTTP, only involve application-level changes in the Web client and server programs. So the choice of platform is not very critical. As we discuss in Chapter 5, our initial implementation was done on the Digital Ultrix and Digital UNIX platforms [26], while a more recent (and different) implementation was done on the BS/OS 3.0 platform.

We present details of our implementation in Chapter 5 through Chapter 9.

3.5 Evaluation

To evaluate our algorithms, we conducted experiments both in the simulator and over real networks. Experiments in the simulator gave us the opportunity to have complete control over the network and to scale up the load on it at will. Experiments over real networks enabled us to validate our simulation results.

The performance metrics we used in our evaluation include:

1. *Throughput*: the amount of useful data transferred in a unit time.
2. *Latency*: the time to complete the transfer of a given size.

3. *Fairness*: the extent of similarity across several instances of a particular metric (e.g., the latency experienced by a set of Web clients). We use the *fairness index* defined in [56] as

$$f = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \sum_{i=1}^{n} x_i^2}$$

where $x_1, x_2, ..., x_n$ are n instances of the metric of interest.

4. Other metrics, such as the number of packet losses, retransmission timeouts and unnecessary retransmissions, that are of interest in specific cases.

3.6 Summary

In this chapter we described our research methodology. The first phase is experimental analysis. We described our experimental testbed and our setup to gather packet traces. The second phase involves designing new algorithms and refining them via simulation. We presented an outline of our simulation environment. The final phase is implementation and performance evaluation. We discussed our implementation platform and evaluation metrics.

In the next chapter, we present an analysis of HTTP performance.