Appendix A

Methodology

In this appendix, I present additional details of the evaluation of Sync-TCP described in Chapter 4. In Section A.1, I discuss decisions made in the design of the network configuration. In Section A.2, I discuss the PackMime web traffic model and characteristics of the traffic generated using the PackMime model. Section A.3 describes the challenges in running simulations involving HTTP traffic, which has components modeled with heavy-tailed distributions. Finally, in Section A.4, I describe the additions I made to *ns* to implement per-flow base RTTs and web traffic generation based on the PackMime traffic model.

A.1 Network Configuration

Initial evaluations of network protocols often use a simple traffic model: a small number of long-lived flows with equal RTTs, data flowing in one direction, ACKs flowing in the opposite direction, and every segment immediately acknowledged. These scenarios may be useful in understanding and illustrating the basic operation of a protocol, but they are not sufficiently complex to be used in making evaluations of the potential performance of one protocol versus another on the Internet, which is an extremely complex system. Networking researchers cannot directly model the Internet, but should use reasonable approximations when designing evaluations of network protocols [PF97].

A.1.1 Evaluating TCP

Allman and Falk offer guidelines for evaluating the performance of a set of changes to TCP, such as Sync-TCP [AF99]. These guidelines include the following :

- *Use delayed ACKs.* A receiver using delayed acknowledgments will wait to see if it has data to transmit to the sender before sending an ACK for received data. If the receiver has data to transmit before the timeout period (usually 100 ms) or before there are two unacknowledged data segments, the ACK is piggy-backed with the outgoing data segment bound for the sender. Delayed ACKs are commonly used in the Internet and should be used in TCP evaluations. **Delayed ACKs are used in the Sync-TCP evaluations.**
- *Set the sender's maximum send window to be large.* A sender's congestion window cannot grow larger than its maximum send window, so to assess the effect of a congestion

control protocol on the sender's congestion window, the congestion window should not be constrained by the maximum send window. The bandwidth-delay product is the maximum amount of data that can be handled by the network in one RTT. **The sender's maximum send window is set to the bandwidth-delay product in the Sync-TCP evaluations.**

- *Test against newer TCP implementations.* TCP Reno is a popular implementation of TCP, but performance of new TCP protocols should be compared against the performance of the latest standards-track TCP implementations, such as TCP with selective acknowledgments (SACK) and Explicit Congestion Notification (ECN). **Sync-TCP is evaluated against TCP Reno and ECN-enabled TCP SACK.**
- *Set router queue sizes to at least the bandwidth-delay product.* The maximum number of packets that a router can buffer affects maximum queuing delay that packets entering the router experience. **The router queue sizes are set to two times the bandwidth-delay product in the Sync-TCP evaluations.**
- *Test with both drop-tail and Random Early Detection (RED) queuing mechanisms.* Although most routers in the Internet use drop-tail queue management, RED (or some variant of RED) active queue management is becoming increasing popular. **Sync-TCP over drop-tail routers is compared to TCP Reno over drop-tail routers and ECN-enabled TCP SACK over Adaptive RED routers.**
- *Use competing traffic.* Most traffic in the Internet is bi-directional (*e.g.*, TCP is inherently two-way). In the case of TCP, data segments in one direction flow alongside ACKs for data flowing in the opposite direction. Therefore, ACKs flow on the same paths and share buffer space in the same queues as data segments. This has several implications [Flo91b, ZSC91]. First, since buffers in routers are typically allocated by IP packets, rather than bytes, an ACK occupies the same amount of buffer space as a data segment. Second, since ACKs can encounter queues, several ACKs could arrive at a router separated in time, be buffered behind other packets, and then be sent out in a burst. The ACKs would then arrive at their destination with a smaller inter-arrival time than they were sent. This phenomena is called *ACK compression*. ACK compression could lead to a sudden increase in a sender's congestion window and a bursty sending rate. Third, since there could be congestion on a flow's ACK path, ACKs could be dropped. Depending on the size of the sender's congestion window, these ACK drops could be misinterpreted as data path drops and retransmissions would be generated. This could be detrimental to a sender, which would lower its sending rate in response to the perceived segment drop. These types of complexities occur in the Internet and, therefore, are useful in simulations. **Two-way competing traffic is used in the Sync-TCP evaluations.**
- *Use realistic traffic.* Much of the traffic in the Internet is HTTP traffic, which typically contains a large number of short-lived flows and a non-negligible number of very long-

	One-Way		Two-Way	
segments	260,218		446,129	
data	260,156	(100%)	229,021	(51%)
ACK	62	(0%)	217,108	(49%)
drops	6181		11,169	
data	6178	(100%)	7106	(64%)
ACK	3	(0%)	4063	(36%)
utilization	99.83%		90.30%	
goodput	9983.11	kbps	8722.81	kbps

Table A.1: Performance of One-Way FTP Traffic and Two-Way FTP Traffic Without Delayed ACKs

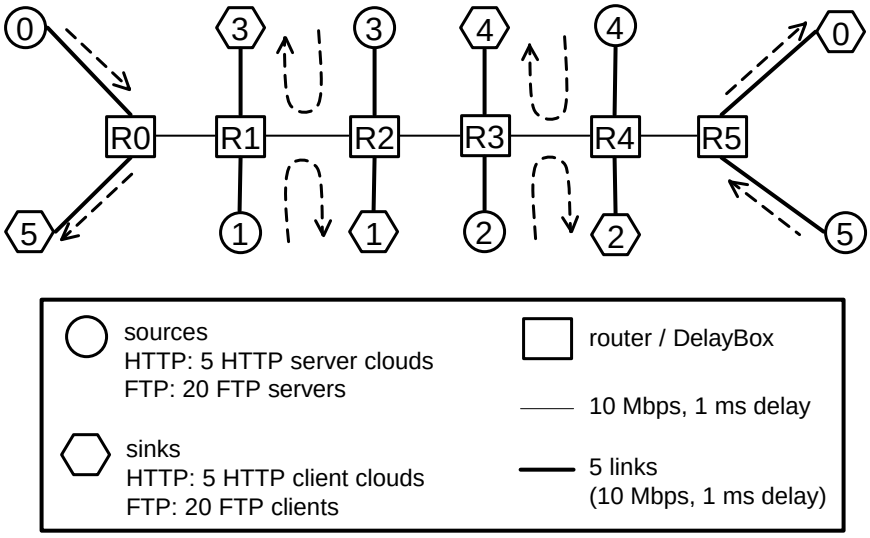

Figure A.1: Simplified 6Q Parking Lot Topology

lived flows. **A recent HTTP traffic model is used to generate web traffic in the Sync-TCP evaluations.**

A.1.2 Impact of Two-Way Traffic and Delayed ACKs

With two-way traffic and no delayed acknowledgments, ACKs make up almost half of all segments sent over the forward path. The addition of ACKs on the forward path increases total segment drops over that of one-way traffic. Table A.1 shows the results of an experiment with 20 FTP bulk-transfers without delayed ACKs flowing in one direction ("one-way") and 20 FTP bulk-transfers flowing in each direction ("two-way") over the parking lot topology (Figure A.1). With two-way traffic, ACKs make up half of the segments sent over the bottleneck link and a significant portion of the total segment drops. Link utilization is lower with two-way traffic than with one-way traffic because ACKs make up half of the packets

	No Delayed ACKs		Delayed ACKs	
segments	446,129		363,553	
data	229,021	(51%)	229,778	(63%)
ACK	217,108	(49%)	133,775	(37%)
drops	11,169		5685	
data	7106	(64%)	4149	(73%)
ACK	4063	(36%)	1537	(27%)
utilization	90.30%		90.59%	
goodput	8722.81	kbps	8868.83	kbps

Table A.2: Performance of Two-Way FTP Traffic Without and Without Delayed ACKs

that the router on the bottleneck link forwards. Thus, two-way traffic creates quantitatively and qualitatively different congestion dynamics and hence is important to understand.

When using two-way traffic, ACKs for reverse-path data segments compete with forward-path data segments for space in the bottleneck router's queue. Using delayed acknowledgments reduces the frequency of ACKs, limiting them to an average of one ACK for every two segments received. Table A.2 compares the results from 20 two-way FTP bulk-transfers without delayed ACKs (as described above) with 20 two-way FTP bulk-transfers with delayed ACKs (the "TCP Reno" experiments described in Chapter 4). On a congested link, using delayed ACKs lowers the number of packets drops without reducing goodput as compared to an ACK-per-packet policy.

A.2 HTTP Traffic Generation

The web traffic generated in the evaluation of Sync-TCP is based on a model of HTTP traffic, called PackMime, developed at Bell Labs [CCLS01b]. In this section, I discuss PackMime, characteristics of the traffic generated by PackMime, and how PackMime is used to generate programmable levels of congestion. Later, in section A.4.1, I will describe the implementation of PackMime in *ns*.

A.2.1 PackMime

PackMime is based on the analysis of HTTP 1.0 connections from a trace of a 100 Mbps Ethernet link connecting an enterprise network of approximately 3,000 hosts to the Internet [CLS00, CCLS01a]. HTTP 1.0 requires a separate TCP connection for each HTTP request-response pair. In other words, only one HTTP request and its response are carried in a TCP connection. The fundamental parameter of PackMime is the HTTP connection initiation rate (which, for HTTP 1.0, is also the TCP connection initiation rate). The PackMime model also includes distributions of base RTTs, the size of HTTP requests, and the size of HTTP responses.

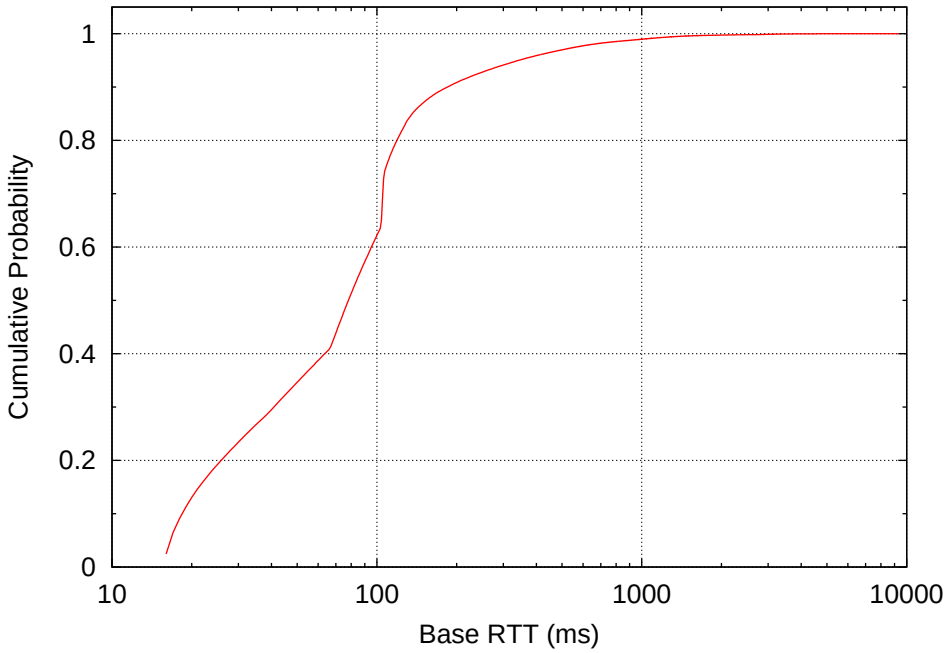

Figure A.2: HTTP Round-Trip Times

PackMime specifies when new HTTP requests should be made. When the time comes for a new request, the request size and server for the client to contact are chosen according to the distributions in PackMime. When a server receives a request, the response size and the server delay time are chosen. Once the server delay time has passed (simulating the server processing the request), the server will transmit the response back to the client.

The RTTs generated by the PackMime model range from 1 ms to 9 seconds. The topology used in the Sync-TCP evaluation (Figure A.1) adds additional propagation delay of 14 ms for end-to-end flows (1 ms on each link in each direction). With the added propagation delay, the base RTTs range from 15 ms to 9 seconds. The mean RTT is 115 ms, and the median RTT is 78 ms. The CDF of the RTTs is presented in Figure A.2.

Both the request size distribution and the response size distribution are heavy-tailed. Figure A.3 shows the request size CDF for 85% load. There are a large number of small request sizes and a few very large request sizes. Over 90% of the requests are under 1 KB and fit in a single segment. The largest request is over 1 MB.

Figures A.3 and A.4 show the CDF and CCDF of the response sizes for 85% load. In the PackMime model, the body of the response size distribution is described empirically, and the tail is a Pareto ($\alpha = 1.23$) distribution. (The Pareto distribution is an example of a heavy-tailed distribution.) About 60% of the responses fit into one 1420-byte segment, and 90% of the responses fit into 10 segments, yet the largest response size is almost 50 MB. Using these

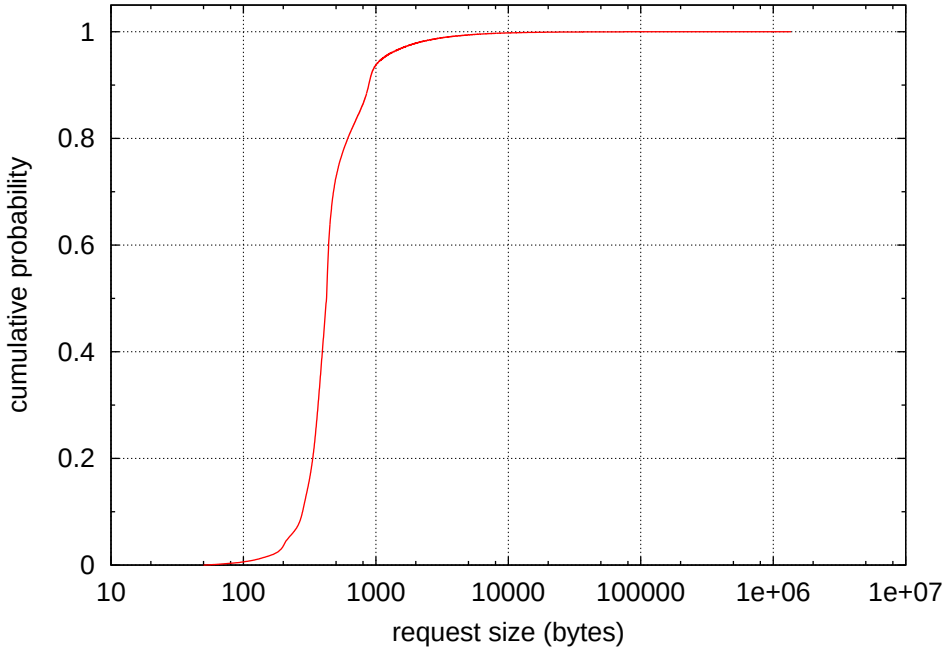

Figure A.3: CDF of HTTP Request Sizes

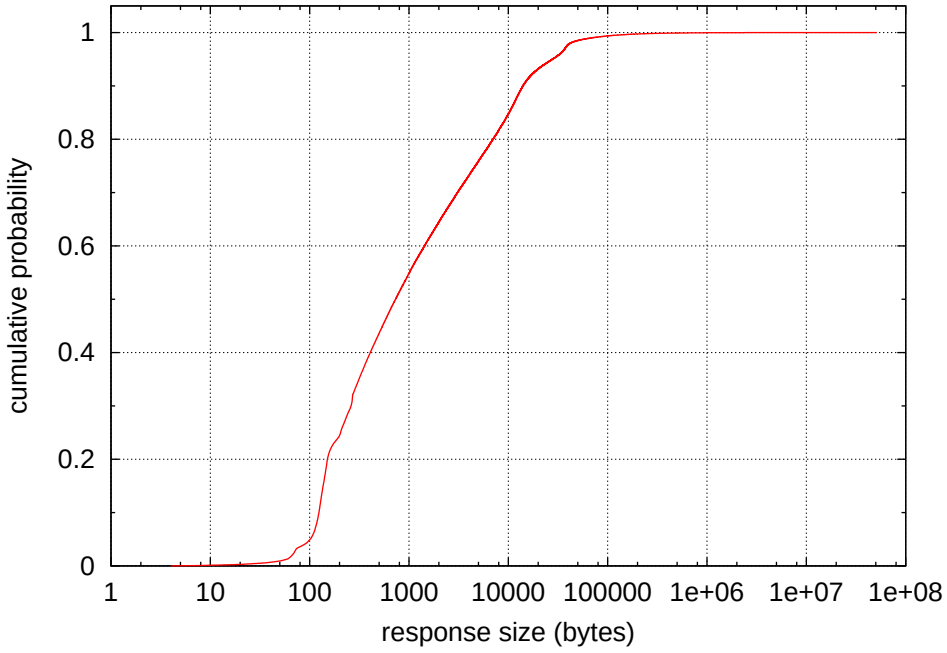

Figure A.4: CDF of HTTP Response Sizes

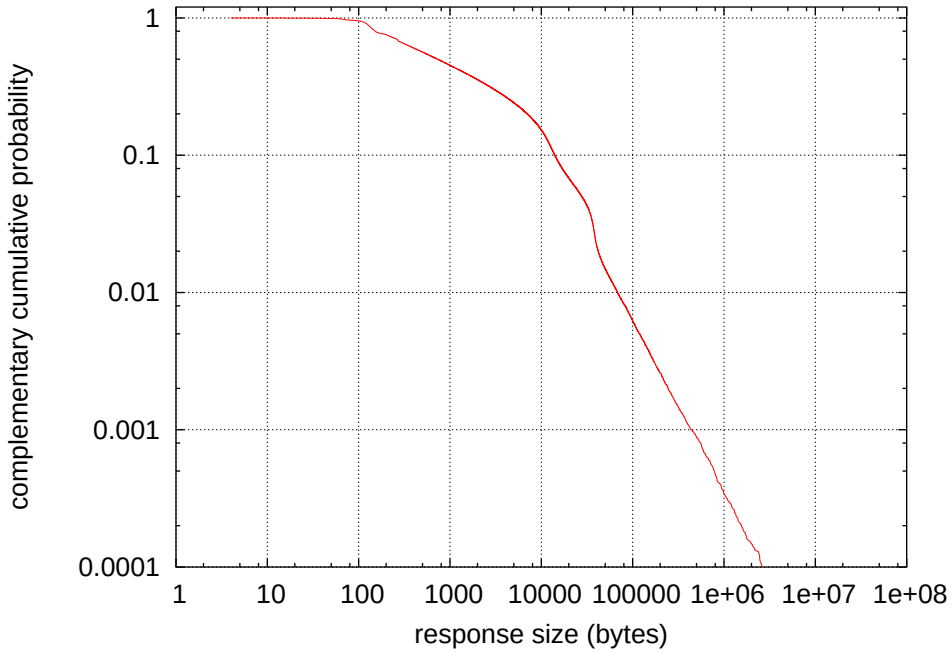

Figure A.5: CCDF of HTTP Response Sizes

distributions to generate web traffic will result in many short-lived transfers, but also some very long-lived flows.

I implemented PackMime traffic generation in *ns* using Full-TCP, which includes bi-directional TCP connections, TCP connection setup, TCP connection teardown, and variable segment sizes. The experimental network consists of a number of PackMime "clouds." Each PackMime cloud represents a group of HTTP clients or servers. The traffic load produced by each PackMime cloud is driven by the user-supplied connection rate parameter, which is the average number of new connections starting per second, or the average HTTP request rate. The connection rate corresponding to each desired link loading was determined by a calibration procedure described below. New connections begin at their appointed time, whether or not any previous connection has completed. Thus, the number of new connections generated per second does not vary with protocol or level of congestion, but only with the specified rate of new connections. For illustration, in Figure A.6, I show the number of new connections per second started by five PackMime clouds for a resulting average load of 85% of the 10 Mbps bottleneck. This was generated by specifying a connection rate of 24.1 new connections per second on each of the five PackMime clouds.

A.2.2 Levels of Offered Load

The levels of offered load used in the evaluation of Sync-TCP are expressed as a percentage of the capacity of a 10 Mbps link. I initially ran the network with all links configured at 100

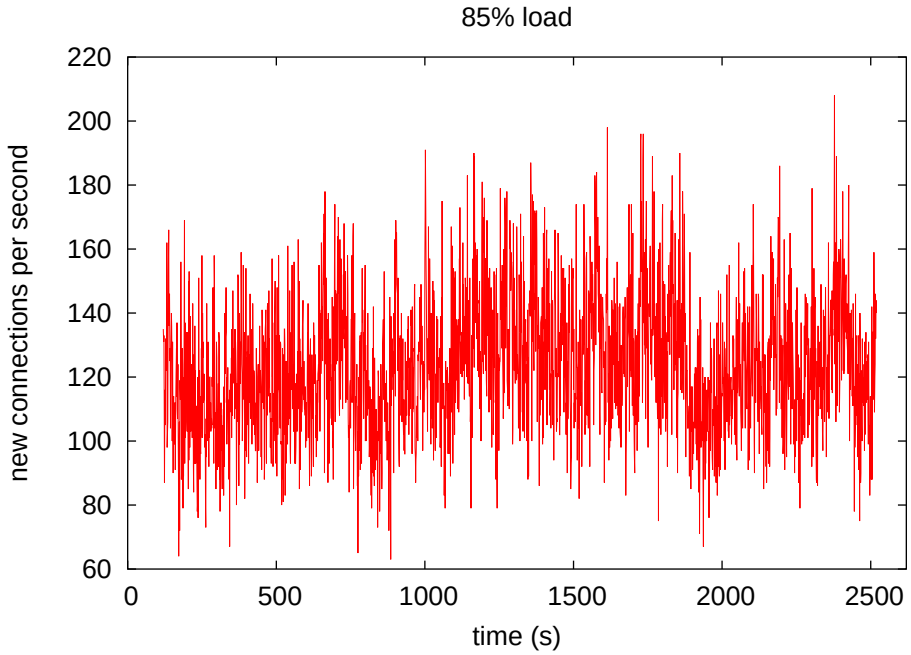

Figure A.6: New Connections Started Per Second

Mbps (for an uncongested network) to determine the PackMime connection rates that would result in average link utilizations (in both forward and reverse directions) of 5, 6, 7, 8, 8.5, 9, 9.5, 10, and 10.5 Mbps. The connection rate that results in an average utilization of 8.5% of the uncongested 100 Mbps link will be used to generate an offered load on the 10 Mbps link of 8.5 Mbps, or 85% of the 10 Mbps link. Note that this "85% load" (*i.e.*, the connection rate that results in 8.5 Mbps of traffic on the 100 Mbps link) will not actually result in 8.5 Mbps of traffic on the 10 Mbps link. The bursty HTTP sources will cause congestion on the 10 Mbps link, and the actual utilization of the link will be a function of the protocol and router queue management scheme used. Note, also, that this 8.5 Mbps of traffic on the uncongested link is an average over a long period of time. For example, Figure A.7 shows the offered load per second at the 85% level on an unconstrained network. Notice that just before time 1000, there is a large burst in the offered load. This burst is typical of HTTP traffic and is included in the average load that results in 8.5 Mbps. This is just an illustration that although I will refer to 85% load, the load level as time passes is not uniform. The burst in the offered load corresponds to a transfer of a 4.5 MB file followed about 30 seconds later by a 49 MB file. Figure A.8 shows the same 85% load on a constrained 10 Mbps network. After the large transfer begins, the number of bytes transmitted per second increases greatly and does not drop below 6 Mbps until the transfer completes around time 1600.

Table A.3 shows the HTTP connection rates per PackMime cloud required to generate the desired load on the forward and reverse paths. Note that these connection rates are dependent

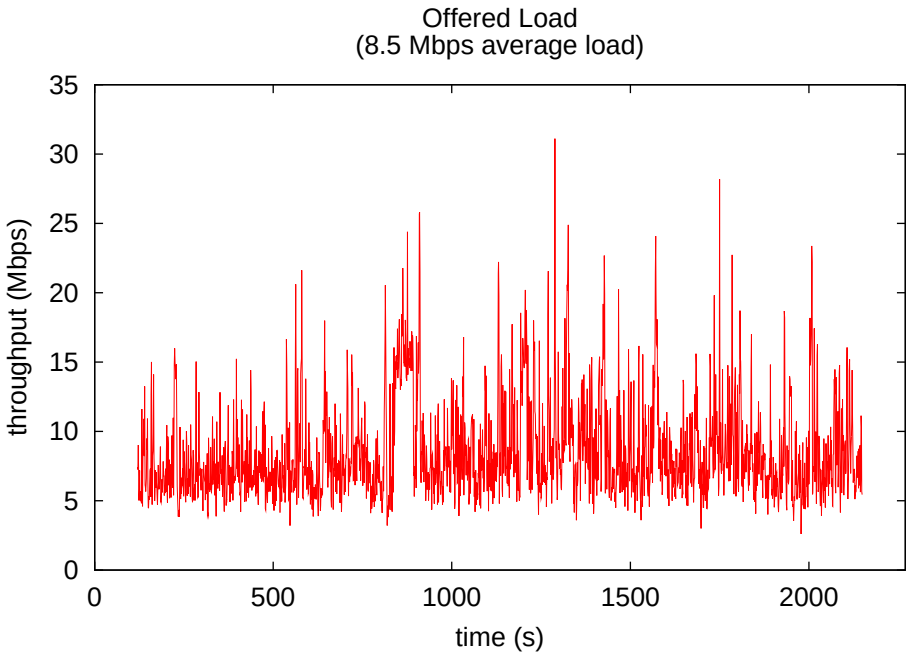

Figure A.7: Offered Load Per Second

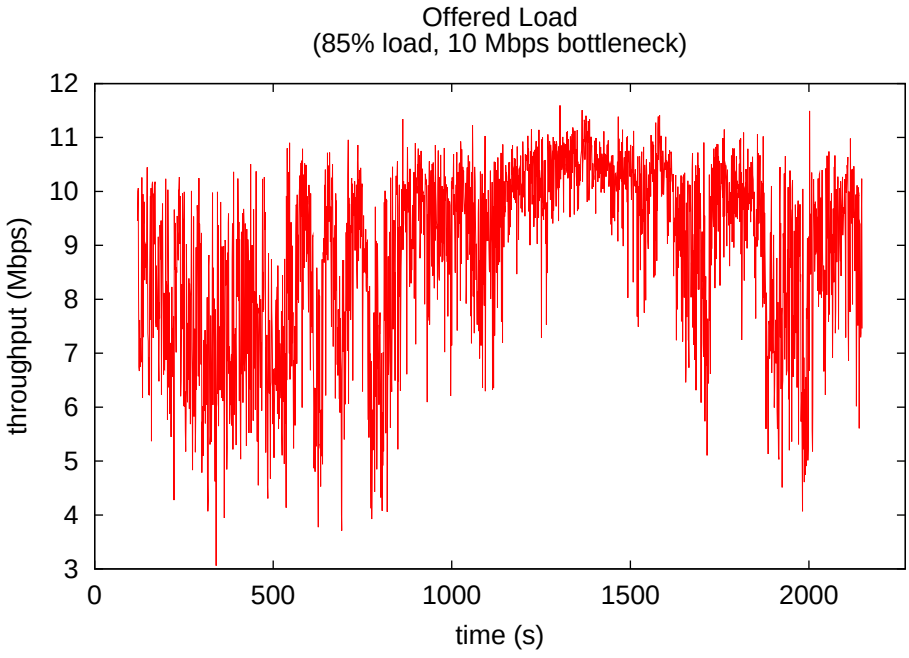

Figure A.8: Offered Load Per Second on a Congested Network

% Load	Forward Path c/s	Reverse Path c/s	Duration (s)
50	14.2	13	3600
60	16.9	15.7	3300
70	19.9	18.7	3000
80	22.7	21.5	2400
85	24.1	22.9	2400
90	25.4	24.2	2100
95	26.9	25.7	2100
100	28.2	27	1920
105	29.6	28.4	1800

Table A.3: HTTP End-to-end Calibration

	75% Total Load		90% Total Load		105% Total Load	
End-to-End % Load	Link 12 c/s	Link 34 c/s	Link 12 c/s	Link 34 c/s	Link 12 c/s	Link 34 c/s
50	7.7	7.2	11.5	11.4	15.3	15.5
60	4.7	4.1	8.7	8.6	12.7	12.5
70	1.5	1.3	6.3	5.7	10.2	9.8
80	-	-	3.3	2.7	7.7	7.2
85	-	-	1.5	1.3	6.3	5.7
90	-	-	-	-	4.7	4.1
95	-	-	-	-	3.3	2.7
100	-	-	-	-	1.5	1.3
105	-	-	-	-	-	-

Table A.4: HTTP Cross-Traffic Calibration

upon the seed given to the pseudo-random number generator. Using a different seed would cause different requests and response sizes to be generated at different times which would affect the offered load. The table also gives the simulation length needed for 250,000 HTTP request-response pairs to complete. The simulation run length will be discussed further in section A.3.

For the multiple bottleneck HTTP experiments, the amount of cross-traffic to generate also had to be calibrated. Table A.4 shows the forward-path connection rates that correspond to the desired load of cross-traffic.

A.2.3 Characteristics of Generated HTTP Traffic

Previous studies have shown that the size of files transferred over HTTP is heavy-tailed [BC98, Mah97, PKC96, WP98]. With heavy-tailed file sizes, there are a large number of small files and a non-negligible number of extremely large files. These studies have also shown that the arrival pattern of web traffic is self-similar, which exhibits long-range dependence, or

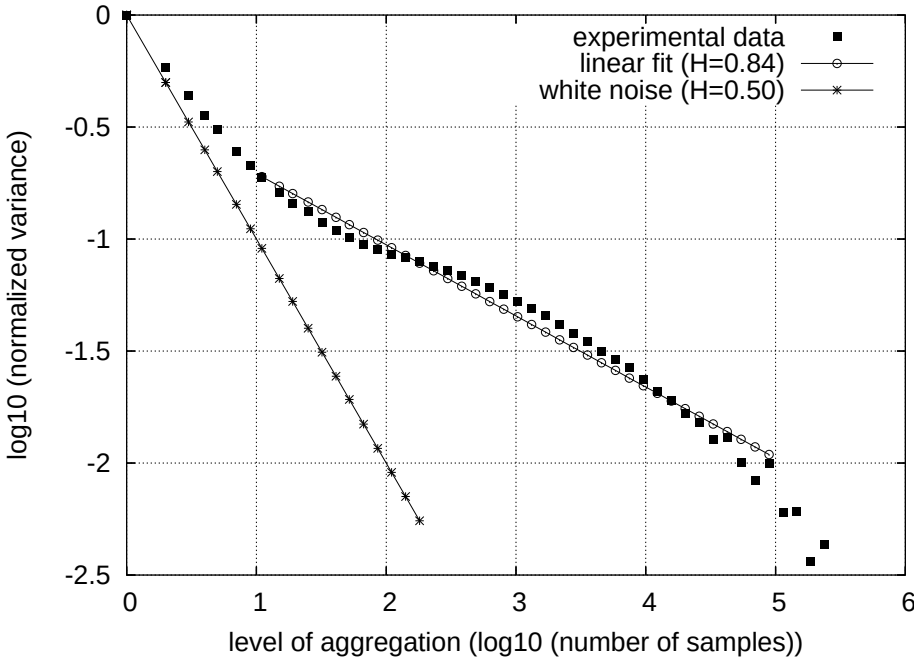

Figure A.9: Packet Arrivals

burstiness over several time scales. In other words, as the aggregation interval increases, the rate of packet arrivals are not smoothed out (as it would be, for example, if the arrivals were Poisson). Figure A.9 demonstrates the long-range dependence of the packet arrivals in the PackMime model[1]. The Hurst parameter characterizes the amount of long-range dependence in a time series. A Hurst parameter of 0.5 is white noise, meaning that there is no long-range dependence, while a Hurst parameter greater than 0.8 means that there is strong long-range dependence. The PackMime model exhibits long-range dependence with a Hurst parameter of 0.84.

A.3 Simulation Run Time

Typically, simulations are run for as long as it takes the sample metric of interest to reach within a certain range of the long-run metric. I am interested in looking at throughput, goodput, and HTTP response times over many HTTP connections. *Throughput* is defined as the amount of data per second sent from a sender to a receiver. I measure throughput by counting the number of bytes (both data and ACKs) transferred every second on the link before the bottleneck router. Because throughput includes dropped segments and retransmissions, it may exceed the bottleneck link bandwidth. *Goodput* is defined as the amount of data per second received by a receiver from a sender. I measure goodput by counting the number of

[1]The data in the figure was computed using methods developed by Taqqu [Taq].

data bytes that are received by each receiver every second. Goodput measures only the segments that are actually received and so does not include segment losses. On an uncongested network, the goodput is roughly equal to the throughput minus any ACKs. *Response times* are application-level measurements of the time between the sending of a HTTP request and the receipt of the full HTTP response.

To assess appropriate simulation length, I will run my experiments on an uncongested network and so will look at only throughput and response times (since goodput is approximately the same as throughput in the uncongested case). A major component of both throughput and response time is the size of the HTTP response. The larger a response, the more data a server tries to put on the network in a small amount of time, which will increase throughput. As response sizes grow, so do the number of segments needed to carry the data, which increases response times.

The PackMime model, used for HTTP traffic generation, has a HTTP response size distribution where the body is described empirically and the tail is a Pareto ($\alpha = 1.23$) distribution. The Pareto distribution is an example of a heavy-tailed distribution, where there are many instances of small values and a few, though non-negligible, instances of very large values. Pareto distributions with $1 \leq \alpha \leq 2$, such as the PackMime response size tail, have infinite variance, meaning that the variance never converges even with very large sample sizes. Crovella and Lipsky show that when sampling from heavy-tailed distributions it can take a very long time for statistics such as the mean to reach steady state [CL97]. For example, to achieve two-digit accuracy for the mean of a Pareto distribution with $\alpha = 1.23$, over 10^{10} samples are required[2]. Even then, the mean is still rather unstable, because of "swamping" observations that can double the mean with just one sample. The probability that a swamping observation could occur in a simulation using a Pareto ($\alpha = 1.23$) distribution is greater than 1 in 100. For these reasons, Crovella and Lipsky state that when $\alpha < 1.5$, simulation convergence becomes impractical. In determining the run length in my experiments, I look at the HTTP response sizes, so in this case, each sample is a HTTP response size. Assuming that the desired load is 85%, there are about 120 requests per second on the forward path. Assuming that for 120 requests per second there are 120 responses per second, generating 10^{10} HTTP responses would take over 80 million seconds, or more than 2 1/2 years, of simulation time.

Figure A.10 shows the running mean response size for 250,000 samples at 85% load, with the theoretical mean[3] of 7560 bytes marked. (This figure corresponds to the response size CDF and CCDF in Figures A.4 and A.5.) Gathering 250,000 response size samples takes about 40 simulated minutes and six hours in real-time. The sample mean is far from converging to the

[2]The sample mean, A_n, of a Pareto distribution converges as $|A_n - \mu| \sim n^{1/\alpha - 1}$, where n is the number of samples. For k-digit accuracy, $n \geq c\, 10^{\frac{k}{1-1/\alpha}}$. If we let $c = 1$, $k = 2$, and $\alpha = 1.23$, $n \geq 10^{10}$.

[3]The theoretical mean for the PackMime HTTP response size distribution was provided by Jin Cao of Bell Labs.

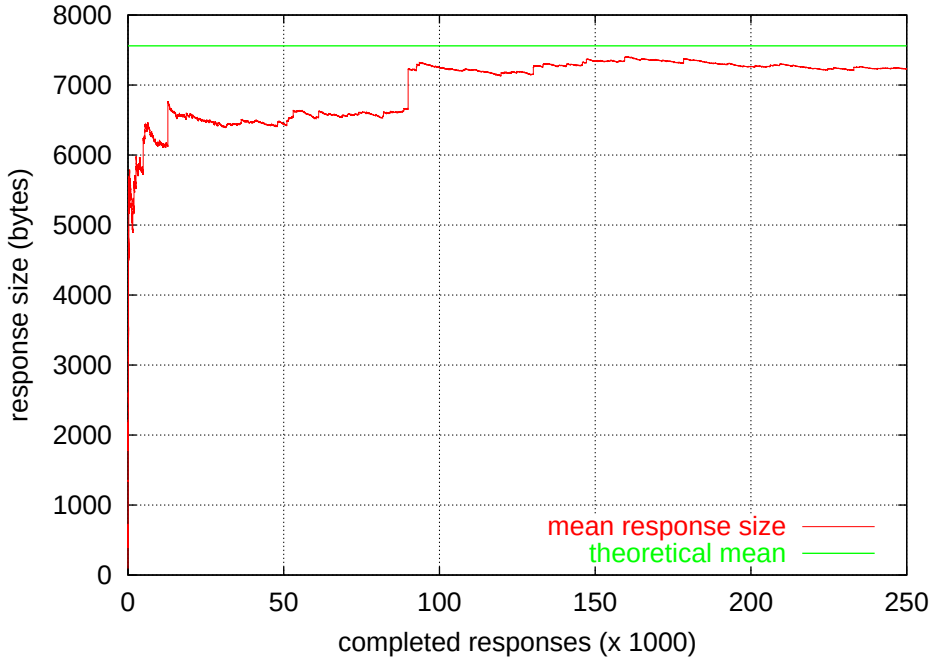

Figure A.10: Mean Response Size

theoretical mean. Therefore, I do not expect to see the mean HTTP response size converge in my simulations.

Since traditional measures like confidence intervals on means will not work, other criteria must be used. Huang offered a suggestion for determining simulation run length: to obtain the 90th quantile of true throughput, run the simulation for at least as long as it takes the 90th quantile of the response size to converge, though no metric for convergence was offered [Hua01]. (By definition, 90% of the values are less than the 90th quantile.) Figure A.11 shows the running 90th quantiles of HTTP response sizes for 250,000 completed responses. After 100,000 samples, the range of the 90th quantiles is less than 100 bytes. To determine how much the 90th quantile is converging, I computed the coefficient of variation (COV). The COV describes how much the standard deviation of the metric, in this case, the 90th quantile of response size, is changing in relation to the mean. Figure A.12 shows the COV of the 90th quantiles of response size[4]. A COV of 5% or less is considered to be stabilizing. The COV of the 90th quantile of response size remains under 3% after 100,000 samples, so at this point, the 90th quantile of response size is converging.

I could also look at the mean and quantiles of throughput and response time, since these are the metrics I am actually interested in measuring. It is a "rule of thumb" in networking research that workloads based on heavy-tailed distributions produce long-range dependent traffic [Par00]. This long-range dependence will appear in the throughput and response

[4] $COV = \frac{\sigma}{\mu}$. The COV is computed every 100 responses.

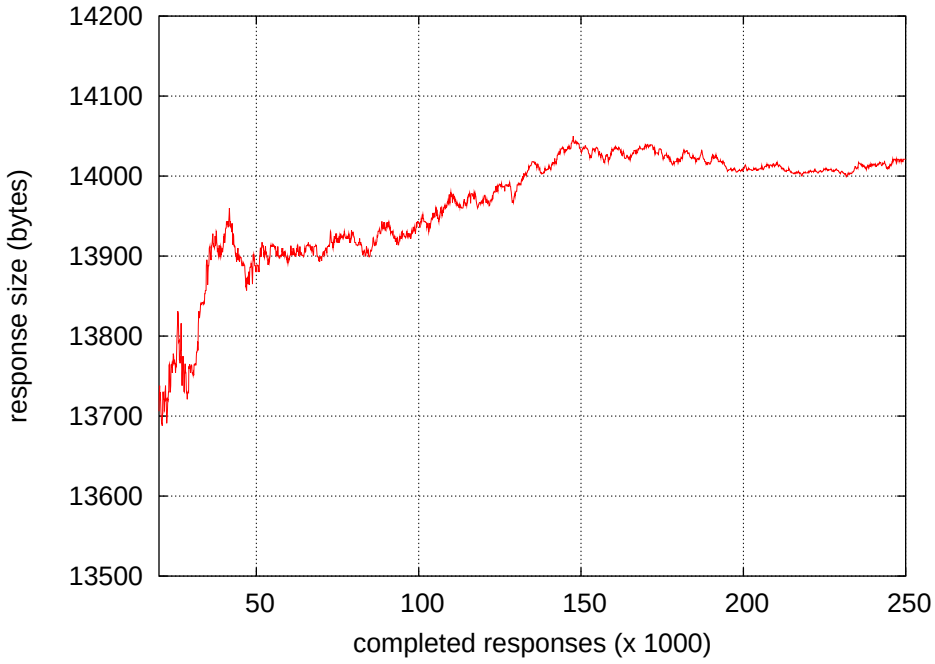

Figure A.11: 90th Quantiles of Response Size

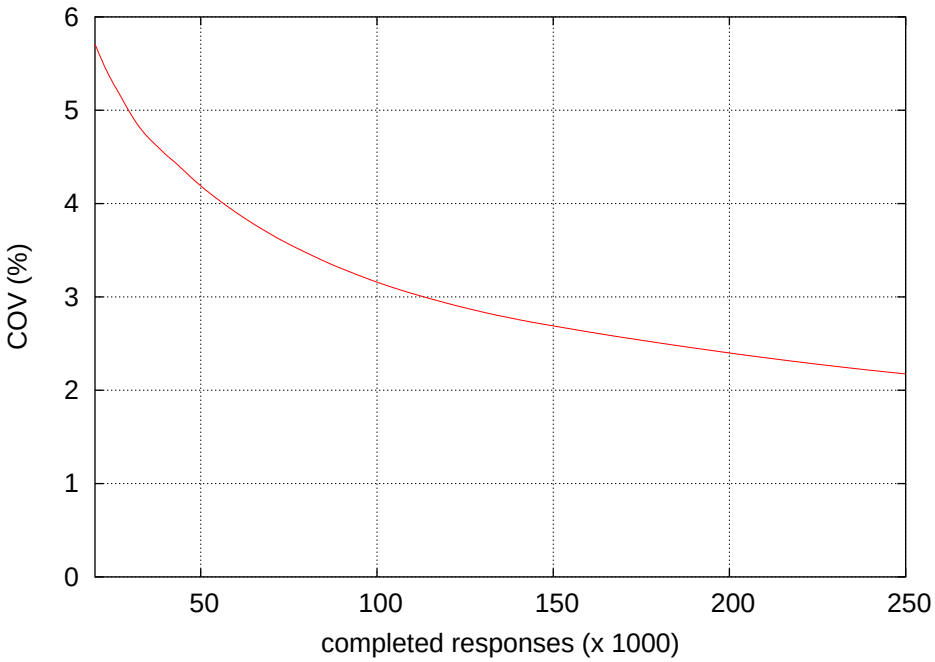

Figure A.12: Coefficient of Variation of 90th Quantiles of Response Size

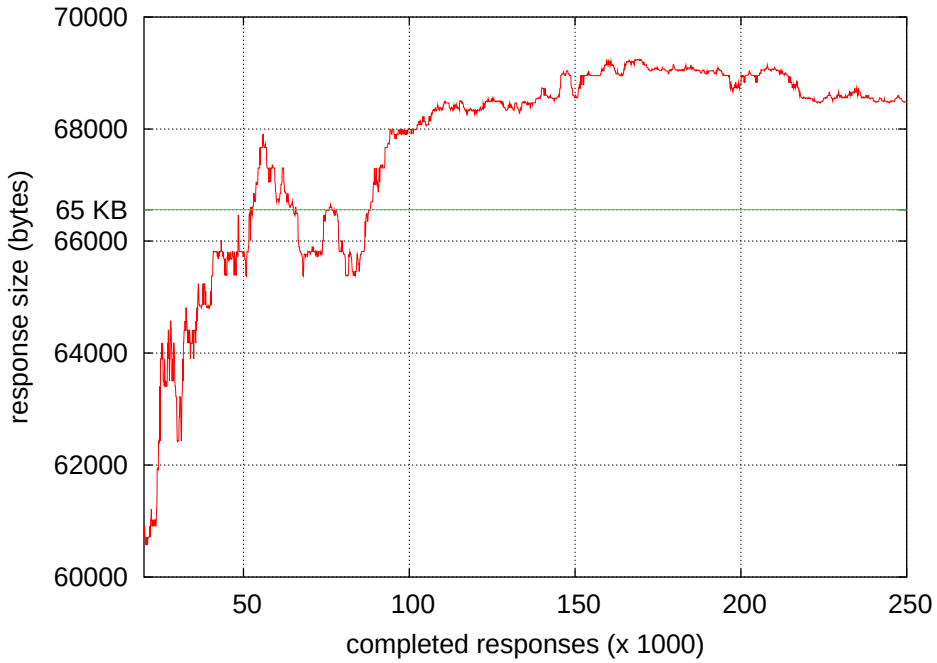

Figure A.13: 99th Quantiles of Response Size

time data. Whereas the response size values are independently sampled from a distribution, throughput and response times are measured values which exhibit dependence. Further, these metrics are protocol- and network-dependent. If I choose the simulation run length based upon throughput or response time, whenever the network setup changed, the simulation run length would have to be re-evaluated.

Operationally, I want to run the simulations long enough to see a mix of "typical" HTTP traffic. This includes small transfers as well as very large transfers. The longer the simulation runs, the more large transfers complete. The sizes of the largest responses can be determined by looking at the 99th quantile of response size. Figure A.13 shows the 99th quantiles of response size for 85% load. After 100,000 completed responses, 1% of all responses are greater than 66,560 bytes (65 KB). This may not seem like a very large response, but it is almost two orders of magnitude larger than the median response size, which is about 730 bytes.

From the 90th quantile of response size heuristic and the large response size heuristic, it looks to be sufficient to run the simulation for 100,000 responses. I will be more conservative and require that 250,000 responses be completed (when CPU and memory resources are abundant for simulation). To ensure that each run has a wide range of response sizes, I also require that the 99th quantile of response size be greater than 65 KB. Also, the maximum-sized response that has begun transfer should be greater than 10 MB. This maximum-sized response is not required to complete, because with heavy loads of traffic there will be much congestion and many packet drops. With the congestion, this maximum-sized response may

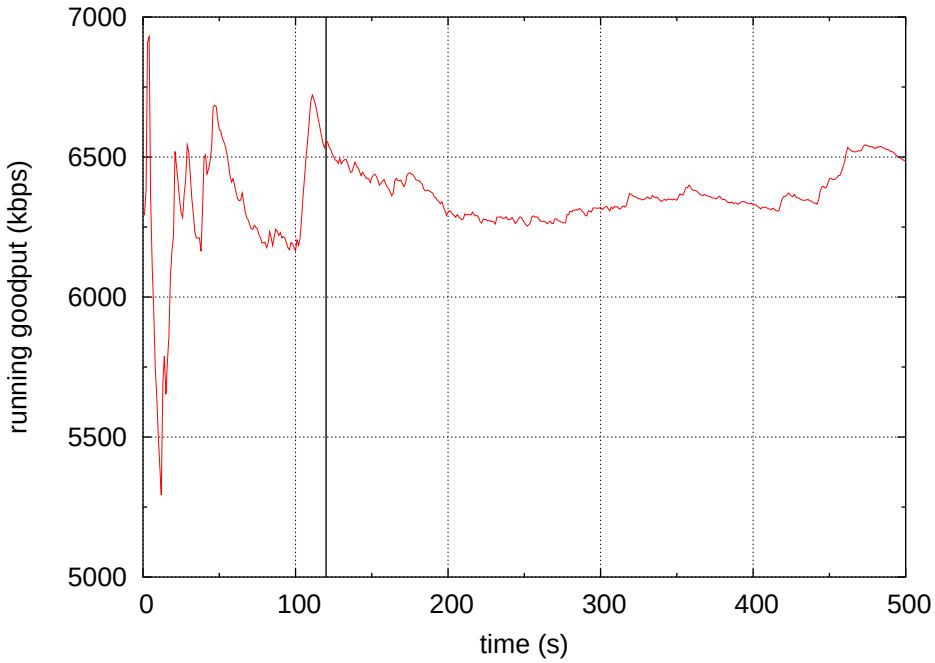

Figure A.14: HTTP Warmup Interval

not complete before 250,000 other responses complete. Even though the maximum-sized response does not complete, its transfer still contributes to the congestion in the network.

In some situations, especially with heavy levels of congestion and multiple bottlenecks, completing 250,000 request-response pairs takes a very long running time (over 6 hours in real-time) and a very large amount of memory (over 500 MB). In these situations, only 100,000-150,000 completed request-response pairs will be required. These numbers of completed request-response pairs still meet the requirements of the heuristics described above.

Before collecting any data in the simulation, including counting completed request-response pairs, a warmup interval of 120 seconds was used. Figure A.14 shows the running goodput of HTTP traffic at 85% load on an uncongested network. The vertical line at time 120 marks the beginning of data collection for the experiments. The running goodput after this point is much less bursty than before time 120.

Ending the simulation before the statistics reach steady-state leaves the HTTP traffic in a transient state. When evaluating the performance of a protocol, I will always compare it against a baseline, which will have been run in the same transient state. When comparing two protocols head-to-head, I will use the same pseudo-random number seeds. This will ensure that the request sizes, response sizes, request generation rate, and link propagation delays remain constant between the two variants. Thus, any observed difference in performance will be isolated to the protocol itself.

A.4 Simulator Additions

To perform the evaluation of Sync-TCP in *ns* with a recent HTTP traffic model, I had to add features not currently available in *ns*. These additions include an implementation of PackMime and a mechanism, called DelayBox, to add per-flow delays and packet drops based on given probability distributions.

A.4.1 PackMime

The PackMime traffic model describes HTTP traffic on a single link between a cloud of nearby web clients and a cloud of distant web servers. Load is specified as the number of new connections that become active per second. A connection consists of a single HTTP 1.0 request and response pair. The PackMime traffic model was developed by analyzing actual traffic traces gathered from a single link between a Bell Labs campus and the Internet. The following connection variables are used to describe traffic in the model:

- *Connection interarrival time* - The time between consecutive HTTP requests.
- *HTTP request size* - The size of the HTTP request that the client sends.
- *HTTP response size* - The size of the HTTP response that the server sends.
- *Server delay* - The time between a server receiving an HTTP request and sending the HTTP response.
- *Server-side transit time* - The round-trip time between a server in the server cloud and the monitored link (located near the client).
- *Client-side transit time* - The round-trip time between a client in the client cloud and the monitored link (located near the client).

PackMime-NS is the *ns* object that drives the simulation of the PackMime traffic model. The network simulator *ns* has objects called *Applications* that control data transfer in a simulation. These Applications communicate via *Agents* which represent the transport layer of the network (*e.g.*, TCP). Each PackMime-NS object is attached to a *ns* node and controls the operation of two types of Applications: PackMime-NS servers and PackMime-NS clients. Each of these Applications is connected to a Full-TCP[5] Agent.

PackMime-NS is meant to be used with DelayBox (to be described in section A.4.2), which controls delay and loss functions. Because DelayBox will control network delay, server-side and client-side transit times are not included in PackMime-NS.

PackMime-NS Implementation

In *ns*, each PackMime cloud is represented by a single *ns* node and is controlled by a single PackMime-NS object. This node can produce and consume multiple HTTP connections at

[5]Full-TCP, as opposed to one-way TCP, includes bi-directional TCP connections, TCP connection setup, TCP connection teardown, and variable segment sizes.

a time. For each connection, PackMime-NS creates new server and client Applications and their associated TCP Agents. After setting up and starting each connection, PackMime-NS schedules the time for the next new connection to begin.

PackMime-NS handles the re-use of Applications and Agents that have finished their data transfer. There are five *ns* object pools used to cache used Application and Agent objects – one pool for inactive TCP Agents and one pool each for active and inactive client and server Applications. The pools for active Applications ensure that all active Applications are destroyed when the simulation is finished. Active TCP Agents do not need to be placed in a pool because each active Application contains a pointer to its associated TCP Agent. New objects are only created when there are no Agents or Applications available in the inactive pools.

For each connection, PackMime-NS creates (or allocates from the inactive pool) two TCP Agents and then sets up a connection between the Agents. PackMime-NS creates (or allocates from the inactive pools) server and client Applications, attaches an Agent to each, and starts the Applications. Finally, PackMime-NS schedules a time for the next new connection to start.

PackMime-NS Client Application A PackMime-NS client Application controls the request size of the transfer. Its order of operations is as follows:

- Sample the HTTP request size from the distribution.
- Send the request to the server.
- Listen for and receive the HTTP response from the server.
- When the response is complete, stop.

PackMime-NS Server Application A PackMime-NS server Application controls the response size of the transfer. Its order of operations is as follows:

- Listen for an HTTP request from a client.
- Sample the HTTP response size from the distribution.
- Sample server delay time from the distribution and set a timer.
- After the timer expires (*i.e.*, the server delay time has passed), send the HTTP response back to the client.
- Once the response has been sent, stop.

PackMime Random Variables

This implementation of PackMime-NS provides several *ns* Random Variable objects for specifying distributions of PackMime connection variables. The implementations were taken from code provided by Bell Labs and modified to fit into the existing *ns* Random Variable framework. This allows PackMime connection variables to be specified by any type of

ns Random Variable, which now includes the PackMime-specific Random Variables. The PackMime-specific Random Variable syntax is as follows:

- `RandomVariable/PackMimeFlowArrive <rate>`
- `RandomVariable/PackMimeFileSize <rate> <type>`
- `RandomVariable/PackMimeXmit <rate> <type>`, where `<type>` is 0 for client-side and 1 for server-side (to be used with DelayBox instead of PackMime-NS).

PackMime-NS Commands

PackMime-NS takes several commands that can be specified in the TCL simulation script:

- `set-client <node>` - Associate the node with the PackMime client cloud.
- `set-server <node>` - Associate the node with the PackMime server cloud.
- `set-rate <float>` - Specifies the average number of new connections per second.
- `set-flow_arrive <RandomVariable>` - Specifies the connection interarrival time distribution.
- `set-req_size <RandomVariable>` - Specifies the HTTP request size distribution.
- `set-rsp_size <RandomVariable>` - Specifies the HTTP response size distribution.
- `set-server_delay <RandomVariable>` - Specifies the server delay distribution.

A.4.2 DelayBox

DelayBox is a *ns* node that is placed in between source and destination nodes and can also be used as a router. With DelayBox, segments from a flow can be delayed, dropped, and forced through a bottleneck link before being passed on to the next node. DelayBox requires the use of Full-TCP flows that are assigned unique flow IDs.

DelayBox maintains two tables: a rule table and a flow table. Entries in the rule table are added by the user in the TCL simulation script and specify how flows from a source to a destination should be treated. The fields in the rule table are as follows:

- source node
- destination node
- delay Random Variable (in ms)
- loss rate Random Variable (in fraction of segments dropped)
- bottleneck link speed Random Variable (in Mbps)

The loss rate and bottleneck link speed fields are optional.

Entries in the flow table are created internally and specify exactly how segments from each flow should be handled upon entering the DelayBox node. The flow table's values are obtained by sampling from the distributions given in the rule table. The fields in the flow table are as follows:

- source node
- destination node
- flow ID
- delay (in ms)
- loss rate (in fraction of segments dropped)
- bottleneck link speed (in Mbps)

Flows are defined as beginning at the receipt of the first SYN of a new flow ID and ending after the sending of the first FIN. Segments after the first FIN are not delayed or dropped.

DelayBox also maintains a set of queues to handle delaying segments. There is one queue per entry in the flow table (*i.e.*, the queues are allocated per-flow). These queues are implemented as delta queues, meaning that the actual time to transmit the segment is kept only for the segment at the head of the queue. All other segments are stored with the difference between the time they should be transmitted and the time the previous segment in the queue should be transmitted. The actual time the first bit of the packet at the tail of the queue should be transmitted is stored in the variable *deltasum*, named so because it is the sum of all delta values in the queue (including the head segment's transfer time). This value is used in computing the delta for new packets entering the queue. If the bottleneck link speed has been specified for the flow, a processing delay is computed for each segment by dividing the size of the segment by the flow's specified bottleneck link speed. The propagation delay due to the bottleneck link speed is also factored into the new packet's delta value.

When a segment is received by a DelayBox node, its transfer time (current time + delay) is calculated. (This transfer time is the time that the first bit of the segment will begin transfer. Segments that wait in the queue behind this segment must be delayed by the amount of time to transfer all bits of the segment over the bottleneck link.) There are two scenarios to consider in deciding how to set the segment's delta:

- If the segment is due to be transferred before the last bit of the last segment in the queue, its delta (the time between transferring the previous segment and transferring this segment) is set to the previous segment's processing delay. This segment has to queue behind the previous segment, but will be ready to be transmitted as soon as the previous segment has completed its transfer.
- If the segment is due to be transferred after the last bit of the last segment in the queue, its delta is difference between its transfer time and the previous segment's transfer time.

If the current segment is the only segment in the queue, DelayBox schedules a timer for the transfer of the segment. When this timer expires, DelayBox will pass the segment on to the standard packet forwarder for processing. Once a segment has been passed on, DelayBox will look for the next segment in the queue to be processed and schedule a timer for its transfer. All segments, both data and ACKs, are delayed in this manner. Segments that should be

dropped are neither queued nor passed on. All segments in a queue are from the same flow and are delayed the same amount (except for delays due to segment size) and are dropped with the same probability.

In the evaluation of Sync-TCP, no loss rate or bottleneck link speed was used. Losses in the evaluation were only caused by congestion on the actual bottleneck link.

DelayBox Commands

DelayBox takes several commands that can be specified in the TCL simulation script:

- `add-rule <src node id> <dst node id> <delay Random Variable> <loss rate Random Variable> <bottleneck link speed RandomVariable>`
  - Add a rule to the rule table, specifying delay, loss rate, and bottleneck link speed for segments flowing from *src* to *dst*.
- `add-rule <src node id> <dst node id> <delay Random Variable> <loss rate Random Variable>`
  - Add a rule to the rule table, specifying delay and loss rate for segments flowing from *src* to *dst*.
- `list-rules` - List all rules in the rule table.
- `list-flows` - List all flows in the flow table.