tcplib: A Library of TCP Internetwork Traffic Characteristics

Peter B. Danzig Sugih Jamin

Computer Science Department, University of Southern California,
Los Angeles, California 90089-0781

traffic@excalibur.usc.edu

USC-CS-91-495

Abstract

This paper describes *tcplib*, a workload or *source* library for network simulation. This paper motivates the need for tools like *tcplib* and discusses how to incorporate it into a network simulator. *Tcplib* is available by anonymous ftp[1].

1. Introduction

When simulating computer networks, it is necessary to specify the traffic between network nodes. Typically, simulation studies of wide-area TCP/IP networks model traffic as a combination of Poisson processes and maximal rate streams—corresponding to TELNET traffic and large file transfers respectively. Such traffic models are justified when the modeler wants to show, for example, that his flow control or gateway scheduling algorithm responds well to worst case traffic or when essentially nothing is known about the real network traffic. Such traffic models do not reveal how similarly robust algorithms respond to the common case load.

This paper describes *tcplib*, a library to help generate realistic TCP/IP network traffic. *Tcplib* is motivated by our observation that present-day wide-area TCP/IP traffic cannot be accurately modeled with simple analytical expressions, but instead requires a combination of detailed knowledge of the end-user applications responsible for the traffic and certain measured probability distributions as reported in [Cáceres91] and [Danzig92]. We collected three–day traces of wide area Internet traffic at UC Berkeley, University of Southern California, and Bell Communications Research. Our study identified that out of more than 35 different application programs, FTP, SMTP, NNTP, VMNET, TELNET, and RLOGIN are responsible for 96% of wide-area TCP/IP bytes. Two related studies, one at University College London and the other at Lawrence Berkeley Laboratory, identified a subset of these six applications as responsible for most of their wide–area TCP traffic [Crowcroft91] [Paxson91]. *Tcplib* models five of these six applications. We excluded VMNET, an IBM mail exchange application, because it was absent from three of the five traces. Furthermore, since TELNET and RLOGIN have essentially the same characteristics, we have included in *tcplib* only routines describing TELNET's. Additionally, we included characteristics of phone conversations based on the study reported in [Brady65] and a distribution of conversations composition breakdown derived from several stub-network traces. Table 1 lists the routine names included with version 0.9 of *tcplib*.

2. How to Use the Library

We define a *conversation* as the set of TCP connections established by a particular application program. Conversations are bidirectional. We differentiate between interactive and bulk-transfer conversations: the transmission rate and packet size of interactive applications are limited by the users' activity, whereas those of bulk-transfer applications are limited by the network's *maximum transfer unit* (MTU) and flow control. Thus for interactive applications, we have the following characteristics: packet size, packet interarrival time, and duration of conversation. Bulk-transfer applications are characterized by the total amount of bytes transferred.

Applications' characteristics alone are not enough to drive a simulation; we also need to specify the arrival time of each conversation. We will look at per application's characteristics in this and the next section, deferring discussion on arrival time to section 4. It should be noted that we use the Unix system library random number generation routines *drand48()* and *lrand48()* in *tcplib*, thus the users should call the initialization routine *srand48()* with the appropriate random number seed, before calling any of *tcplib*'s routines.

[1]The source library for TELNET, FTP, SMTP, and NNTP is available over anonymous ftp from *jerico.usc.edu* (128.125.51.6) in directory *~ftp/pub/jamin/tcplib*.

To simulate a TELNET conversation, we need to set up its source and destination. The source sends one-byte packets at *telnet_interarrival()* time for *telnet_duration()* of the conversation. The destination waits for packets from the source and sends response packets whose sizes are obtained from *telnet_pktsize()*. This is different from the uniform, unidirectional model of TELNET sources used in previous simulation studies.

Application Type	Routine Name

Interactive	`float telnet_duration()`
	`float telnet_interarrival()`
	`int telnet_pktsize()`
	
	`float phone_talkspurt()`
	`float phone_pause()`

Bulk Transfer	`int ftp_nitems()`
	`int nntp_nitems()`
	
	`int ftp_itemsize()`
	`int nntp_itemsize()`
	`int smtp_itemsize()`
	
	`int ftp_ctlsize()`

Traffic Breakdown	`struct brkdn_dist *`
	`brkdn_dist()`
	
	`char*`
	`next_app(struct brkdn_dist *brkdn_dist)`

Table 1: Distribution functions included in version 0.9 of the traffic library.

To simulate an FTP conversation, we need to, once again, setup its source and destination. The source of an FTP conversation models the number of items sent by calling *ftp_nitems()*. The size of each item is obtained from *ftp_itemsize()*. Each item should be fragmented into network-MTU-size packets followed by a final smaller packet, as needed. Currently we do not model the duration and number of control exchanges between items; however, the sizes of packets sent during this period is available from *ftp_ctlsize()*. On receiving an FTP control packet, the destination of the FTP conversation should response with control packets, also obtained from *ftp_ctlsize()*. Note that our traces rarely found FTP conversations that both sent and received files, but the control channel handshaking does make FTP conversations bidirectional.

An SMTP source sends a mail message after a series of address verification messages. *Tcplib* does not include the distribution of the number of these messages. This may be added in a future release. The SMTP source calls the routine *smtp_itemsize()* to pick an item size. The item size returned represents both the mail message and any address verification messages. Because the response from the destination is negligible, *tcplib* does not model it.

NNTP sources are very similar to FTP sources. Unlike FTP however, NNTP conversations send items in both directions. The current implementation of *tcplib* does not model the bidirectional nature of NNTP properly. Furthermore, since we do not have the distribution of NNTP control packet sizes, we use packets with uniformly distributed sizes smaller than NNTP_ARTICLE_SIZE. We define NNTP_ARTICLE_SIZE to be 250 bytes [Lapsley91].

Phone conversations share a lot of similarities with interactive applications. One party initializes the conversation, then the two parties exchange data, interleaved with pauses, with an occasional cross-talk, and finally the conversation is terminated. *Tcplib* provides the routine *phone_talkspurt()* and *phone_pause()* which return the talkspurt and pause duration respectively. These numbers represent wall-clock time of talkspurts and pauses. In using these routines, we need to multiply the values returned with the data rate of the specific voice-encoding mechanism simulated (e.g. 64kbits/s). The talkspurts and pauses distribution used by *tcplib* were gathered from the study reported in [Brady65].

3. Random Number Generation

Because curve fitting loses information and since it makes no difference to the simulator whether there exists an analytical representation of the distributions, *tcplib* generates random numbers by the inverse transform method [Jain91]. It inverts a piecewise linear representation of the measured distribution. Below, we briefly describe the inverse transform method.

The inverse transform method maps uniformly distributed 0–1 random variates through the "y-axis" of the cumulative probability distribution onto the "x-axis." With distributions fitted to analytical expressions, the inverse transform method involves inverting an equation. Consider, for example, generating an exponential random variate. If μ is a 0–1 uniform random variate and l is the parameter of the exponential, then $x=-log(1-\mu)/l$ is an exponentially distributed random variate. In our case, we built a histogram of the individual data points, and then summed the histogram bin heights to create our distribution function. Hence, our distributions are represented by arrays rather than expressions. An array index i corresponds to a particular value of the distribution. The contents of the array element at index i, $x[i]$, is the value of the cumulative distribution. Hence, to generate a random variate, we first generate a 0–1 uniform random variate μ. We then perform a binary search on the array elements until we find the element $x[k]$ into which μ falls. Finally, we linearly interpolate between $x(k)$ and $x(k+1)$ to determine our random variate x.

Another approach we could have adopted is to keep a sorted set of every single data point: instead of building a histogram of the data points and search for the bin in which the random variate μ fell, we would create an array of 100 or 1000 elements, corresponding to 0.01-quantile and 0.001-quantile increments, indexed directly by μ. We diecided against this approach because it takes more memory to implement than does the scheme described in the previous paragraph.

4. Conversation Arrival Times and Traffic Breakdown

Given the characteristics that completely describe each application's conversations, we now need to decide when to start a new conversation. Molina showed in [Molina27] that phone conversations are served after an exponentially distributed delay from the time of their arrivals. To the network, the interarrival times of phone conversations are thus exponentially distributed. When simulating telephone conversations, a function similar to the one in Fig. 1 could be used to determine the arrival time of the next conversation. Notice that the value of λ (mean interarrival) in Figure 1 depends on the time-of-day and the activity level of the site simulated.

```
double
phone_interarrival()
{
   return((-(double)λ) * log(drand48()));
}
```

Fig. 1: Exponential interarrival time.

Unfortunately, we do not have such a description for arrival times of data applications. *Tcplib* assumes an exponential interarrival time for all conversations. For each arrival time, we then decide which application to assign it to by calling the *next_app()* routine. *Next_app()* takes as an argument a traffic breakdown distribution obtained from the routine *brkdn_dist()*. *Brkdn_dist()* takes the mean (η) and variance (σ^2) of each application's probability of occurrence and returns a traffic breakdown distribution. The mean and variance of an application's occurrence probability were calculated from its measured breakdown in our traces. In generating the traffic breakdown distribution, *brkdn_dist()* uses the Ahrens method for generating random numbers distributed as a gamma distribution of integer order r, the r-stage Erlangian distribution. Gamma distribution has a low enough coefficient of variation to produce random numbers with variance matching our measurements. Equations 1 and 2 give the gamma density function and the standard deviation of Erlangian distribution taken from [Press88] and [Kleinrock75] respectively.

$$p_r(x)dx = \frac{x^{r-1} e^{-x}}{\Gamma(r)} dx, \qquad x > 0 \qquad \text{(Eq. 1)}$$

$$\sigma_{Erlang} = \frac{1}{\sqrt{r}}\left(\frac{1}{\eta}\right) \qquad \text{(Eq. 2)}$$

5. Compiling *tcplib*

Recall that characterizing an application can require several distributions. For example, TELNET needs library routines for TELNET conversation duration, packet interarrival time, and packet size (See Table 1). We collected all the distribution needed to create the library from our trace of U.C. Berkeley traffic, and placed them in ASCII files. You will find these files in directory *data*.

The program *tcplibgenh* reads each of the distribution files and translates them into initialized C language arrays. These arrays contain the cumulative probability distribution for each application characteristic, as described in the previous section. The arrays for each application are placed in a C language header file. For example, *telnet.h* contains the initialized arrays for duration, packet interarrival time, and packet size.

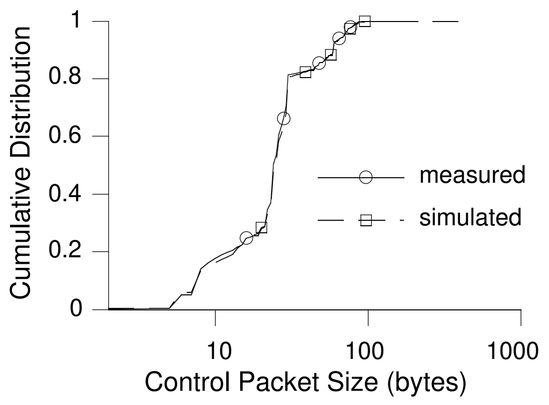
As explained in the previous section, *tcplib* generates a uniform 0–1 random variate and finds the cumulative distribution bin in which the variate falls. It then performs a linear interpolation between the lower and upper bounds of the bin's contents to find the value of the random variate.

The program *tcplibgenc* creates the procedures for generating application characteristics, the ones you link with. *Tcplibgenc* also adds the names of these procedures to the file *tcpapps.h.*

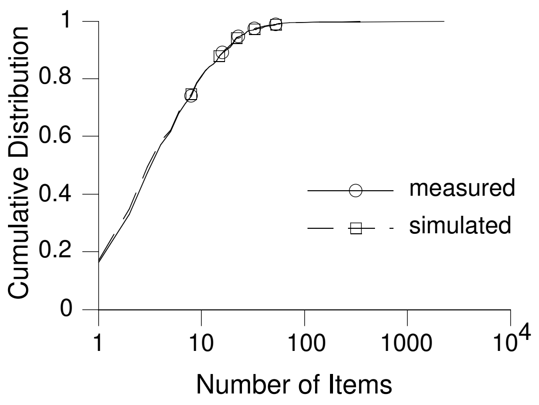
Similarly, *breakdown.c* contains the program that reads in a set of traffic breakdown data, and generates *app_brkdn.h*. *App_brkdn.h* contains an array with the mean and variance of each application's breakdown. *Brkdn_dist.c* contains the routine *breakdown()* which uses the array in *app_brkdn.h* to generate a stub network's traffic breakdown. *Brkdn_dist.c* also contains the routine *next_app()*, which takes a stub network's traffic breakdown and randomly generates the name of an application. To use the traffic breakdown routines in *tcplib*, you need to include the file *brkdn_dist.h* in your source code.

6. Testing *tcplib*

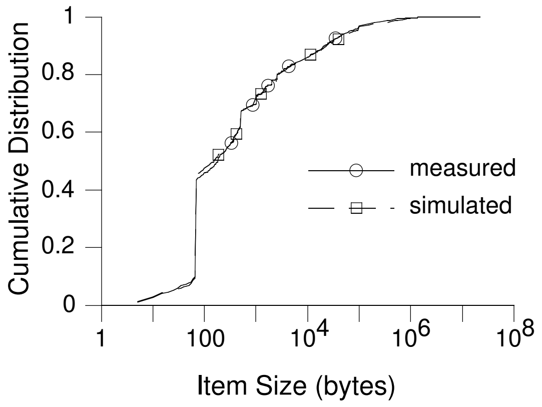
To test the library we generated one thousand data points for each application characteristic. Figure 2 compares the cumulative probability distribution of the generated data points with those from the original traces. The curves match very well. Note that this does not test correlation in the random processes; this is another area for future work.



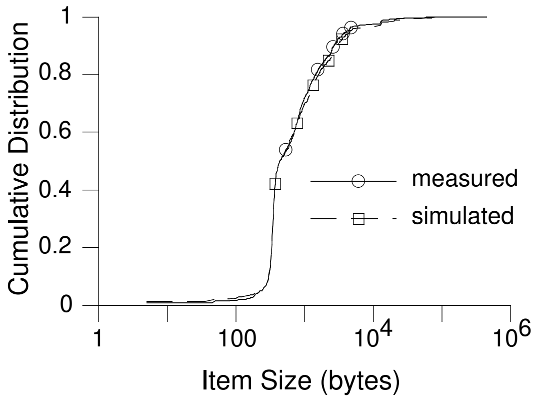
(a) FTP control packet sizes



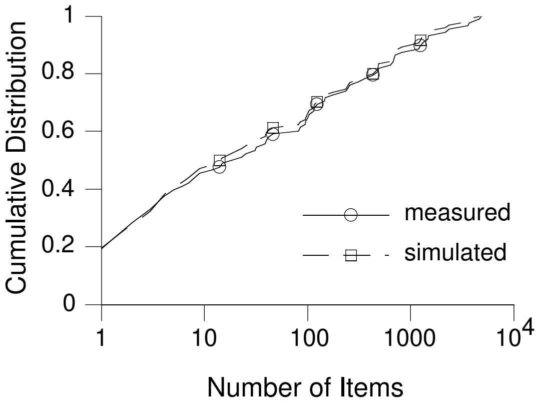
(b) Number of items sent per FTP conversation



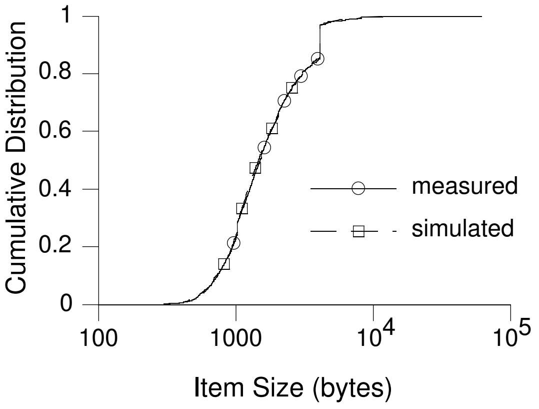
(c) FTP item size



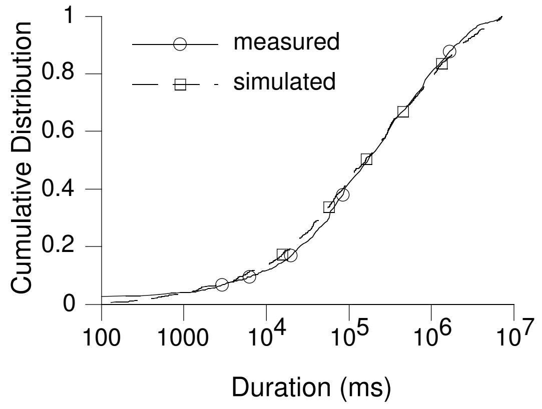
(d) SMTP item size



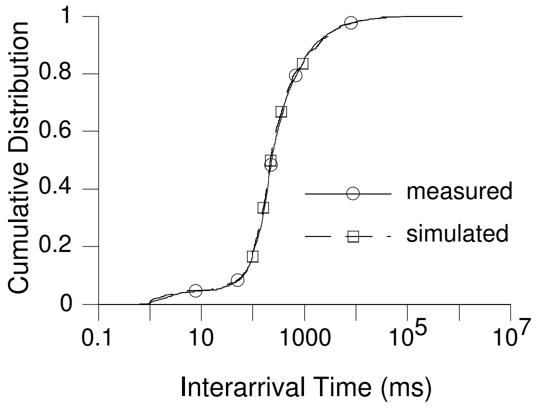
(e) Number of items sent per NNTP conversation



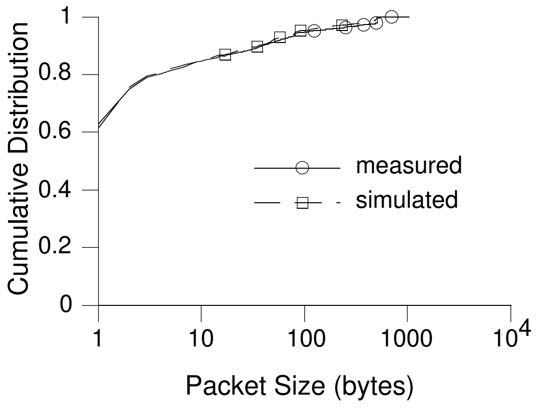
(f) NNTP item size



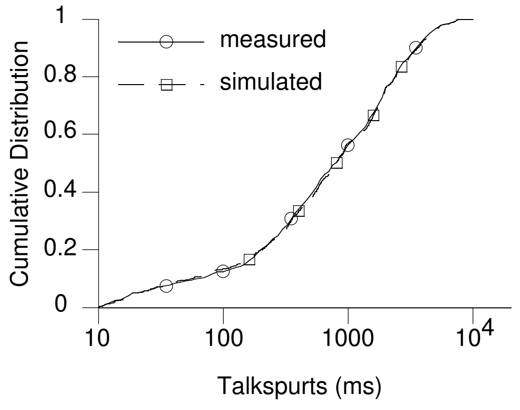
(g) Duration of TELNET conversations



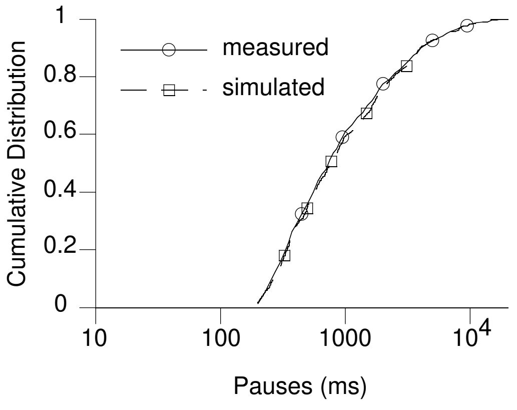
(h) Interarrival times of TELNET packets



(i) TELNET packet sizes



(j) Phone talkspurts



(k) Phone pauses

Fig. 2: Comparison of measured and simulated characteristics cumulative distribution.

6. Wide-Area Network Simulation

We have incorporated *tcplib* into the *tcpsim* network simulator. *Tcpsim* [McCanne91] is a successor to the REAL network simulator [Keshav88]. Both *tcpsim* and REAL run on top of the discrete-event simulator NEST [Dupuy90]. In this section we describe extensions and modifications we made to *tcpsim* to support our workload model. We hope that in describing the changes we made to *tcpsim* to accommodate *tcplib*, we will give the readers an inclination on how to incorporate *tcplib* into their simulators.

6.1. Representing Conversations and Conversation Arrivals

We modified *tcpsim*'s simulation description language to allow specification of stub-networks' conversation interarrival time. Given a stub-network's conversations' mean interarrival time, we use *tcplib* to generate conversation arrival times. To each arriving conversation, we need to assign a simulator thread. Ideally, such threads could be dynamically created and destroyed. Unfortunately, *tcpsim* only supports static allocation of simulator threads. We therefore need to know the number of threads needed by each application. For this, we added a "dry-run" mode to *tcpsim*. At the end of a dry-run, the modified *tcpsim* prints out the simulation topology and the required number of threads per application for the specified simulation duration.

We observed that not all conversations of a given application overlap in simulation time. Thus we could specify a number of thread less than the number of conversation arrivals. A thread that has finished simulating a conversation could be assigned to the next conversation. This scheme works fine except when the gateway queueing mechanism, such as Fair-Queueing [Demers89] or Virtual Clock [Zhang90], uses the conversation identification (*cid*), which is associated with each thread, to regulate its use of the gateway's buffer space. Since it is non-trivial to modify *tcpsim* to generate a new *cid* for a given thread, we have run our simulations with the same number of threads as the total number of conversations.[2] As a thread finishes simulating a conversation, it is put to sleep by scheduling it at a time later than the simulation duration. Because NEST uses a long-integer format for passing the next run-time of a thread to the scheduler, we had problems with long-running simulations and had to modify the scheduler to take an argument in the float format.[3]

Finally, we should also note the change we made to the configuration language to allow us to describe a number of threads representing similar conversations in one specification.

6.2. Application-driven Data Transfer with Retransmission

The next modification we made to *tcpsim* regulates the sending of application data. Originally, the transport layer of *tcpsim* calls the application layer for more data to send whenever the simulated network interface is free. The data sizes of each application is fixed by a specification in the simulation configuration file at startup time. Our changes allow each application to control its packet sizes and transmission times.

Since applications running under the original *tcpsim* are assumed to use one-size packets, *tcpsim* did not keep transmitted packets around for retransmission. We, however, wanted the number of packets and bytes sent to conform to our measured data, so we added a retransmission queue to *tcpsim*.

6.3. Bidirectionality of Data Flow

The final important change we made to *tcpsim* allows bidirectional traffic flow. In the original design of *tcpsim*, only the source of a conversation runs a transport protocol. The destination of a conversation runs a *sink* routine which does nothing but sends ACK packets. We incorporated the *sink* routine to the transport protocol and run the same protocol on both the source and destination of a conversation. On receiving a packet, the transport protocol sends an ACK packet back to the sender and interrupts the application layer, notifying it of an incoming packet. The application layer might choose to transmit some data in response.

7. Limitations and Future Work

Tcplib needs to be extended in several ways. Most crucially, it needs a better model of conversation arrival rates. In [Danzig92] we showed that these arrival rates are site dependent, hence this will require detailed study from many Internet stub networks. *Tcplib* currently lacks several application specific details. It does not model the interarrival time of FTP control packets. It does not model the distribution of number of request response handshakes that occur during SMTP and NNTP conversations. It does not model TELNET packet sizes longer than the network MTU. It does not model VMNET and other applications, such as video conferencing, that may eventually consume significant wide–area network bandwidth.

The simulator also needs some major overhaul. Routing between nodes is currently determined statically at the beginning of a simulation run. To study routing protocols, such as multi-path routing, it would be necessary to add dynamic routing to the simulator. We need to modify *tcpsim* to support thread re-use and/or dynamic thread creation. *Tcpsim* could also use some more extensible statistics gathering and tracing mechanisms. Currently it does only a minimal amount of statistics gathering, preferring to print out event traces for later analysis. Considering the amount of resident memory required to run the simulation, and the flexibility of not having to re-run the simulation for every new set of statistics one becomes interested in, deferring statistics gathering to a later phase is a very good decomposition. Unfortunately, simulation of 1000 seconds simulation time could easily produce a 150 megabyte trace file. And finally, and most crucially, we need to replace the event generation mechanism used by NEST. NEST uses the UNIX *signal()* system call to activate each and every event in the simulation, such that half of the simulator's running time is spent inside the UNIX kernel.

Despite its limitations, we believe that *tcplib* makes an important contribution to performance modeling of TCP/IP internetworks. We are interested in your comments and bug reports.

[2]A newer version of *tcpsim* allows connection setup and teardown, which could potentially be used for thread recycling.
[3]This introduces undefined randomization to timing, a different encoding that doesn't lose precision would be better.

References

[Brady65]	Brady, P.T., "A Technique for Investigating On-Off Patterns of Speech," *The Bell System Technical Journal*, Jan '65, pp. 1-22.
[Cáceres91]	Cáceres, R., Danzig, P.B., Jamin, S., and Mitzel, D.J., "Characteristics of Wide-Area TCP/IP Conversations," *Proc. of ACM SIGCOMM '91*, pp. 101-112.
[Crowcroft91]	Crowcroft, J., *Traffic Analysis of Some UK-US Academic Network Data*, University College London Tech. Rep. RN/90/UK.
[Danzig92]	Danzig, P.B., Jamin, S., Cáceres, R., Mitzel, D.J., Estrin, D., "An Empirical Workload Model for Driving Wide-Area TCP/IP Network Simulations," to appear in the *Journal of Internetworking: Practice and Experiences*, 1992.
[Demers89]	Demers, A., Keshav, S., and Shenker, S., "Analysis and Simulation of a Fair Queueing Algorithm," *ACM SIGCOMM '89*, pp. 2-12.
[Dupuy90]	Dupuy, A., et al., "NEST: A Network Simulation and Prototyping Testbed," *CACM* 33:10, April '90, pp. 63-74.
[Jain91]	Jain, R., *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991.
[Keshav88]	Keshav, S., *REAL: a Network Simulator*, UCB Tech. Report. UCB/CSD 88/472, 1988.
[Kleinrock75]	Kleinrock, L., *Queueing Systems*, vol. 1, John Wiley & Sons, 1976.
[Lapsley91]	Lapsley, P., E-mail communication, 1991.
[McCanne91]	McCanne, S., unreleased code for *tcpsim*, 1991.
[Molina27]	Molina, E.C., "Application of the Theory of Probabilities to Telephone Trunking Problems," *Bell System Tech. Jl.*, 6, 1927, pp. 461-494.
[Paxson91]	Paxson, V., *Measurements and Models of Wide-Area TCP Conversations*, Lawrence Berkeley Lab. Tech. Rep. LBL-30840, 1991.
[Press88]	Press, W.H., et al., *Numerical Recipes in C*, Cambridge University Press, 1988.
[Zhang90]	Zhang, L., "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *Proc. of SIGCOMM '90*, pp. 19-29.