

A USER LEVEL MODEL FOR ARTIFICIAL INTERNET TRAFFIC  
GENERATION

A Thesis Presented To

The Faculty of the

Fritz J. and Dolores H. Russ  
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Issam Safa

November 2000

THIS THESIS ENTITLED  
“A USER LEVEL MODEL FOR ARTIFICIAL INTERNET TRAFFIC  
GENERATION”

by Issam Safa

has been approved

for the School of Electrical Engineering and Computer Science  
and the Russ College of Engineering and Technology

---

Dr. Shawn Ostermann  
Associate Professor of Computer Science

---

Jerrel R. Mitchell, Dean  
Fritz J. and Dolores H. Russ  
College of Engineering and Technology

## ACKNOWLEDGEMENTS

I would like to thank Dr. Shawn Ostermann for allowing me this opportunity to work with him. I would like to acknowledge his patience, encouragement, and assistance in providing thoughtful insights at all the stages of the work. I would also like to thank my labmates, most notably Vitaly, Priya, Ethan, and Brett for their help, and for making me feel at home among them. I want to thank my friends Rahul, Usman, Ahmad, Iyad, Kunjan and Arvind for their support throughout my study at Ohio University. I would also like to thank Eric Helvey for his prompt responses, and the valuable information he provided me. I would like to acknowledge the funding granted to me by the school of Engineering and Computer Science, and finally a special thanks to Radwa for being there at the right time.

## TABLE OF CONTENTS

	Page
LIST OF TABLES . . . . .	vi
LIST OF FIGURES . . . . .	viii
1 INTRODUCTION . . . . .	1
1.1 Motivation for the thesis . . . . .	1
1.2 Transmission Control Protocol (TCP) . . . . .	2
1.3 Network Traffic Emulation . . . . .	4
1.3.1 Analytical vs. Empirical Models . . . . .	5
1.3.2 Packet Level Emulation . . . . .	6
1.3.3 Conversation Level Emulation . . . . .	7
1.4 Existing Software . . . . .	9
1.4.1 TCPlib . . . . .	10
1.4.2 Tcptrace . . . . .	11
1.4.3 TCPDiscardD . . . . .	12
1.4.4 Trafgen . . . . .	12
2 USER LEVEL MODEL . . . . .	1
2.1 Design . . . . .	1
2.1.1 Data Acquisition . . . . .	1
2.1.2 Statistical Processing . . . . .	6
2.1.3 Traffic Generation . . . . .	10
3 MODEL VERIFICATION . . . . .	1
3.1 Seed Data . . . . .	1
3.2 Evaluation Method . . . . .	2

3.2.1	Comparing Data Sets . . . . .	3
3.3	Experimental Procedure . . . . .	5
3.4	Correlations of Artificial Traffic . . . . .	6
3.5	Difficulties in Generating and Correlating Artificial Traffic . . . . .	12
3.5.1	Isolated Network . . . . .	12
3.5.2	<i>tcplib</i> Interpolation . . . . .	13
3.5.3	unidirectional <i>http</i> connections . . . . .	13
3.5.4	User To Application Correlation . . . . .	15
4	CONCLUSIONS AND FUTURE WORK . . . . .	1
4.1	Conclusions . . . . .	1
4.2	Future Work . . . . .	2
	BIBLIOGRAPHY . . . . .	54
	<b>APPENDIX</b>	
A	INVERSE TRANSFORMATION . . . . .	57
A.1	Principle . . . . .	57
A.1.1	Proof . . . . .	57
A.1.2	Example . . . . .	58
B	EXAMPLE DATA FILE . . . . .	59
C	CORRELATION DATA . . . . .	62
	ABSTRACT . . . . .	67

## LIST OF TABLES

Table	Page
1.1 1991 Application Breakdowns . . . . .	8
1.2 1995 Application Packet Counts . . . . .	9
1.3 1997 Application Breakdowns . . . . .	9
3.1 conventions . . . . .	7
3.2 summary . . . . .	7
3.3 Correlation Values for OU Artificial Timed Series . . . . .	8
3.4 Correlation Values for OU Artificial Timed Series . . . . .	8
3.5 Correlation Values for NewWave Artificial Timed Series . . . . .	9
3.6 Correlation for NewWave Artificial Timed Series . . . . .	10
3.7 Correlation for NewWave Artificial Timed Series . . . . .	10
3.8 Correlation for NewWave Artificial Timed Series . . . . .	10
3.9 Correlation for NewWave Artificial Timed Series . . . . .	11
3.10 Correlation for NewWave Artificial Timed Series . . . . .	11
3.11 Correlation for NewWave Artificial Timed Series . . . . .	11

3.12	<i>tcplib</i> Interpolation Results . . . . .	14
C.1	conventions . . . . .	63
C.2	Correlation Values for OU Artificial Timed Series . . . . .	63
C.3	Correlation Values for OU Artificial Timed Series . . . . .	64
C.4	Correlation for OU Artificial Timed Series . . . . .	64
C.5	Correlation for OU Artificial Timed Series . . . . .	64
C.6	Correlation for OU Artificial Timed Series . . . . .	65
C.7	Correlation for OU Artificial Timed Series . . . . .	65
C.8	Correlation for OU Artificial Timed Series . . . . .	65
C.9	Correlation for OU Artificial Timed Series . . . . .	66

## LIST OF FIGURES

Figure	Page
2.1 The TrafGen Model . . . . .	11
2.2 TCPLib Thread Flowchart . . . . .	13
2.3 TELNET Application Thread Flowchart . . . . .	14
2.4 FTP Application Thread Flowchart . . . . .	15
2.5 HTTP Application Thread Flowchart . . . . .	16
2.6 SMTP Application Thread Flowchart . . . . .	17
2.7 The User Model . . . . .	19

## 1. INTRODUCTION

The rapid growth of the Internet and its increased heterogeneity has led to an added complexity in protocol design and testing. This heterogeneity, which ranges from the individual links that carry the network traffic to the protocols that inter-operate over the links to the different applications used at a site, imposes serious challenges to protocols designers. A major difficulty in this area is determining how newly designed and updated protocols impact various network environments under various loads. Such difficulty necessitates the existence of a certain model that can closely mirror real world network traffic without having to wait for a certain phenomenon to occur on a physical network.

This thesis describes a new empirical model to emulate real network traffic starting from the user level down to the application level. This chapter discusses the motivation for this thesis, and the key parameters involved into creating the model as well as the capabilities and limitations of the software upon which this model was implemented. In the following chapters we will give a description of the design and implementation of our new model as well as the experimental results of this model, and the conclusion and future works.

### 1.1 Motivation for the thesis

This work finds its origin in the interest of the U.S. National Air and Space Administration (NASA) to integrate satellite links for use in the global Internet [17]. An essential requirement for this endeavor is to be able to create background traffic with customizable intensity and content. Researchers at Ohio University, as part of their research efforts, designed a tool [13] to generate artificial network traffic based

on observed patterns on real networks, this traffic generator used breakdowns for the different applications used on the network separated by the inter-arrival time separating two consecutive applications. A key question following this approach is, how would this approach deal with traffic scalability. One possibility is to scale the inter-arrival time by the same scaling factor as the traffic, this approach means that to scale the traffic we're making users do more work, in other words multiplying the work a user does by the scaling factor needed. Another approach is to characterize an end user with the necessary parameters, and the question of scalability would be easily solved by adding more users to the desired experimental traffic.

This thesis addresses the second approach. By defining and implementing the necessary parameters characterizing a user, we can meet the full objective: a scalable traffic with customizable intensity.

## 1.2 Transmission Control Protocol (TCP)

The evolution of the Internet has pushed TCP to new limits over a wide variety of IP infrastructures. The Transmission Control Protocol is responsible for 95% of the Internet traffic [29]. TCP is a reliable, connection-oriented protocol. This means that TCP guarantees that the data sent out on the network by some source will arrive at the destination intact. TCP operates on top of the Internet Protocol layer (IP). IP itself is a connection-less, best-effort transport protocol. This means that IP will try its best to get the data from the source to the destination, but without a guarantee. With this picture in mind the TCP/IP suite refers to using TCP on top of IP to provide a reliable service guaranteeing the delivery of the traffic to the destination as long as this destination is reachable. TCP specifications [25] specify the necessary steps to open a TCP connection. Put briefly, the source and destination hosts must perform a handshaking operation to establish the connection and synchronize the hosts [5]. This handshaking involves the source sending the first packet called the source's SYN, when the destination receives the source's SYN, it responds with its

own SYN and an acknowledgment (ACK) of the originating source's SYN. When the source receives the destination's SYN and ACK the connection is established and the source is free to send its data to the destination. Closing a TCP connection requires similar steps.

In order to guarantee that the destination receives the transfer properly, TCP uses positive acknowledgement. This means that the destination is required to acknowledge each packet it receives. Until the ACK for a specific packet has been received, the source cannot assume that the packet has been received by the destination. When no ACK arrives at the source in a predetermined amount of time, the source assumes that the packet did not arrive at the destination, and will retransmit the packet. The time required to send a packet and receive that packet's ACK is defined as one-acknowledgment round trip time (RTT). While TCP performs well on terrestrial links, it has been shown that a single TCP connection does not make full use of available bandwidth when hosts are connected with a satellite channel [16]. This poor performance is due among other things, to interaction of various congestion control and avoidance algorithms embedded in TCP that act to minimize the data loss due to network overload [2].

One congestion control and avoidance algorithm in particular, Slow start [28], is strongly influenced by packet delay. Slow start controls the injection of data into the network by restricting the number of packets that a connection can have in transit at one time. Initially, a connection is restricted to having one unacknowledged packet on the network at a time. Each ACK received increases the number of packets the connection can have in transit by one. As long as the source is receiving the for each packet, this sequence of ACKs will double the data rate with each RTT. In long delay systems such as satellites channels, the amount of time needed to reach optimal bandwidth utilization is much larger than for terrestrial networks [1]. This is due to the fact that it takes longer for ACKs to arrive at the source, which means that the number of packets the source can have in transit does not increase as fast as it could.

The result of the TCP congestion control and avoidance algorithms is a reduction in the efficiency of the bandwidth utilization of the channel [1]. This inefficiency results in more time being required to complete a transaction. In long delay systems, this additional time can become very large. Various researchers have suggested modifications to the congestion control and avoidance algorithms [1][4][3][7][8][12][14].

While the behavior of a single TCP connection over a satellite link is well documented and generally understood, the impact of additional traffic on the overall behavior of long delay networks is still undetermined. Obtaining “live” network data showing how TCP reacts in a high traffic satellite environment has been a difficult task for the following reasons:

- Satellite channels are not the preferred communication method in computer networks, and that’s because of the delay overhead for these systems is so much greater than terrestrial channels.
- Terrestrial links greatly outnumber satellite links.
- The cost of using satellite channels has been prohibitive to widespread general use.

For these reasons, much of the testing of protocols on satellite systems has been performed without the benefit of background traffic, thus reducing researchers’ understanding of how protocols will react when faced with competing traffic for the same network resources.

### **1.3 Network Traffic Emulation**

The consequences of allowing an unfinished protocol to exist on today’s Internet could be catastrophic. It is therefore essential that researchers determine how new protocols perform in an open environment, with multiple active connections and varied load, in other words it is crucial to observe how newly designed protocols react to

competing real world traffic, thus the existence of an emulation tool that can create such background traffic becomes essential.

Emulating network traffic on an isolated network would allow a single machine to generate traffic that exhibits the same characteristics of real world network traffic. This would allow protocol testing in controlled environments, under different load with different traffic patterns. Such possibility provides researchers with a more accurate picture of the protocols' behavior in the real world without tempering with the network's integrity.

The design of an accurate network emulator involves making choices about the key parameters of such emulator. Generally speaking, the higher the details' level, the less efficient the emulator is. Historically emulation in general falls into one of two categories: Analytical or Empirical [23] and it's usually designed as a packet level emulation or conversation level emulation. Packet level emulations are usually more exact, but they are slower and require a significant amount of system resources. On the other hand conversation level emulations focus on an entire connection, and require less data to model and less computations. Conversation level emulations rely on random variables, and gives a statistically equivalent traffic instead of an exact replication of the packets.

### **1.3.1 Analytical vs. Empirical Models**

For our purpose, an analytical model of a random variable is defined as a mathematical description of this variable's distribution [23]. Ideally the model has few bound parameters and no free parameters, in which case it fully predicts the distribution of similar random variables derived from datasets other than the ones used to develop the model. In contrast, an empirical model [6] describes a random variable's distribution based on the observed distribution of an earlier sample of the variable. An empirical model may be predictive but not easy to understand.

There are a number of advantages of an analytical model compared to empirical model for the same random variable

- analytical models are often mathematically tractable, thus are more understandable.
- analytical models are very concise and thus easily communicated.
- with analytical models, different datasets can be easily compared by comparing their values for the model's free parameters.

While having these advantages, a key question is whether an analytical model is capable of fully capturing the essence of network traffic. An empirical model perfectly models the dataset from which it was derived. It has the ability to capture irregularities and spikes, the same thing cannot be said of an analytical model.

For this work we chose the empirical approach for the following reasons:

- we're concerned about unusual phenomenon that can only occur in real world traffic, thus an analytical model eliminating such phenomenon for the sake of simplicity and conciseness is defeating the main purpose of the work.
- There already exists a framework for an empirical model called *tcplib* [6] that has been in use for quiet sometime.
- the essence of the approach itself is to capitalize on existing software and tools to come up with the new model, and these software are based on empirical models already.

Following the choice of the empirical model, we have to choose between two levels of emulations, Packet level emulations, and conversation level emulation.

### 1.3.2 Packet Level Emulation

In packet level emulation, the network is analyzed on a packet-by-packet basis. The key characteristics of packet level emulations are packet inter-arrival times, and

packet sizes. These two pieces of information are enough to create a model for network traffic.

Simplicity is the major advantage of packet level emulators. With packet level emulators, the network is described, at the packet level. There are no connections, and no applications. All the calculations are made to determine the size and arrival times of packets independent of all other packets on the network. Such approach greatly reduces the complexity level. Furthermore, the packet level approach eliminates the need to update the emulator, and this is because at the packet level, emulators see nothing above packets, the addition of new protocols or new applications has no effect on the modeling process as long as the data sets contain packets from these new protocols and applications.

One major disadvantage of packet level models is in ignoring the correlation between successive packets. Each application generate packets that have some particular characteristics, thus forming certain pattern in the overall traffic. For example FTP packets for large file tend to be large in size and last only a short period of time, while telnet packets are short and last over longer periods. A packet level emulator will not be able to dissect such patterns and consequently model the network accurately.

### **1.3.3 Conversation Level Emulation**

The building blocks of a conversation level emulator are conversations. Analyzing network traffic means monitoring all the connections on the network and recording specific information about each connection type. As an example, a TELNET[26] connection requires the following parameters to be known for its modeling: the packet sizes, connection duration, and packet inter-arrival times. In such case the emulator is able to construct an entire TELNET connection with equivalent characteristics to the original one when the generated traffic is viewed as a whole.

With such approach, after analyzing the network traces, we can maintain a pool of information about the observed connections' characteristics. when emulating this network we can generate an infinite number of unique connections using the information

in the pool which are taken from a relatively small number of measured connections.

Conversation level emulations are more complex than packet level emulations, and this is because of the additional effort required to parameterize each type of applications, and each type of protocol studied as well as interpreting the data sets at hand. on the other hand conversation level emulation can be vastly improved over packet level emulation. They make it possible to manipulate the data collected about the network being modeled and thus finely control the output which is beyond the capabilities of a packet level emulator.

A complete conversation level emulator must model all types of traffic including traffic generated by user-level applications. For practical purposes it's sufficient to model only the most dominant applications [6]. Although modeling more applications increase the accuracy of the emulator, it should be noted however that such addition of rarely occurring traffic does not give much accuracy in return when considering the computational overhead added.

In 1991, Danzig [6] suggested that only four TCP-based applications were dominant and consequently we can only focus on these applications to accurately model most networks. Table 1.1 describes the relative frequencies of applications measured at the time of the paper preparation.

Table 1.1 1991 Application Breakdowns. This table presents the percentage of TCP conversations seen by application as reported in 1991 by [6].

Application	TELNET	SMTP	NNTP	FTP
% of Conversations	10.2%	63.3%	< 1%	3.5%

In 1995, Frazer [9] reported that according to the NSFNET researchers, and after monitoring and characterizing the Internet traffic, HTTP had eclipsed FTP as the dominant application on the Internet in terms of packet count for the first time. The application breakdown by packet count is given in Table 1.2

Table 1.2 1995 Application Packet Counts. This table presents the percentage of TCP packets seen by application as reported in 1995 by [9].

Application	HTTP	FTP	NNTP	TELNET	SMTP	DNS
% of Packets	21%	15%	8%	8%	6%	5%

In 1997, Thompson[29] showed that TCP traffic was responsible for 95% of the bytes transferred, 90% of the packets transmitted, and 75% of the conversations monitored on several Internet backbone networks. Unlike Danzig, Thompson[29] reported that there were six major contributors to TCP traffic. These applications and their contributions to network traffic are listed in table 1.3.

Table 1.3 1997 Application Breakdowns. This table presents the percentages of Internet traffic by packet count, byte count and conversation as reported in 1997 by [29].

Application	% Conversations	% Bytes	% Packets
HTTP	75%	75%	70%
TELNET	< 1%	< 1%	1%
FTP	< 1%	5%	3%
NNTP	< 1%	2%	< 1%
SMTP	2%	5%	5%
DNS <sup>1</sup>	18%	1%	3%

The key characteristics for the first four applications were defined and reported in [6] while HTTP characteristics were given by Mah in [18], and the integration of those characteristics was done by Helvey [13].

#### 1.4 Existing Software

The model we developed is based on a set of existing tools, This section provides a description for these tools, highlighting their capabilities and limitations.

These tools include, TCPLib, tcptrace, trafgen, and TCPDiscardD. These software were used as the backbone of our model, a general description is given in the following sections. A more detailed description is given in Chapter 2 when describing the model itself.

#### 1.4.1 TCPLib

TCPLib [6] is an empirical model of wide-area traffic. It models the distribution of the random variables (e.g. bytes transferred, duration) associated with different protocols by using the distributions actually measured for those protocols at an Internet site. To perform such task TCPLib provides a library of functions designed to provide a network simulator with a series of network applications, and to provide characteristics for each application. As designed, TCPLib will determine which application's traffic would appear next on the network. Additionally, once the application is known, TCPLib will supply the necessary characteristics to simulate the application. To accomplish this, TCPLib utilizes data collected from one or many networks that contain characteristics of the application and the relative frequencies of the values of each characteristic. This data is then manipulated to obtain the mean and the variance values. Traffic characteristics are then manufactured using a randomizing function based on the mean and variance data.

One very interesting feature of TCPLib is its ability to merge data from multiple networks into a single data set. This data combination has several advantages:

- it reduces the overall effects of transient conditions. In its statistical manipulations of its data sets, TCPLib takes existing data points and weights the values based on the frequency of occurrence. This means that the effect a singular or low frequency event will be reduced.
- merging the characteristics of multiple networks would ensure that a single network's eccentricities are not exaggerated. However, if a single network's characteristics were all that was required, creating a data set from a single

network is also possible.

As originally packaged, TCPLib was able to fully describe TELNET, FTP, NNTP, SMTP, and phone conversations. Notably absent from this list is HTTP. At the time TCPLib was developed,<sup>2</sup> HTTP had just recently been introduced and was not yet a major factor on the Internet. Any network traffic simulator that disregards the effects of HTTP would be ineffective today.

In addition to lacking HTTP, the original TCPLib also had no mechanism for determining when a new conversation would begin. Without this information, a conversation level simulator would be unaware of when to initiate new connections. This information is also vital for a conversation level emulator. One explanation for the lack of conversation times could be that the simulator was intended to determine what connection was next, and to fully service that particular connection. Only after the original connection was completed was a new connection to be opened. This approach assumes that conversations are contiguous on the network. In order to more accurately model network traffic, a mechanism of determining when a new conversation should begin is essential. Both the HTTP support and the inter-arrival times were added to TCPLib in the work of Helvey [13].

#### **1.4.2 Tcptrace**

Tcptrace [21] is a program designed to accept network trace files from a variety of sources and to perform measurements on them. One of the key features of tcptrace is that it is conversation oriented, allowing a single connection to be monitored for the lifetime of the connection. Tcptrace is able to breakdown files of packets into their corresponding connections, thus making it easier for the emulator to characterize conversations.

Tcptrace can be used to generate the data files needed by TCPLib. Although not originally equipped with the tools to generate these files, tcptrace did have the

---

<sup>2</sup>TCPLib was developed in 1991 at the University of Southern California.

proper framework to allow a module to be integrated into tcptrace that creates the data files[13].

### 1.4.3 TCPDiscardD

TCPDiscardD [22] is a program designed to allow connections from remote hosts, accept any data transferred to it, and to discard the data. This functionality is needed by a traffic generator in order to provide a destination for artificial network traffic.

Although many UNIX operating systems are distributed with a discard server, these servers have been augmented with various security mechanisms. These security options include disabling the server if it is invoked frequently, or refusing connections if a particular host attempts to connect to the server too often. In an emulation environment, these security featuring destroy the utility of the discard server. This is the main reason for choosing a user-level server.

TCPDiscardD was originally designed to be used on a single port, but a newer versions allow connections to an arbitrary number of ports. This expansion allowed the output of the traffic generator to be monitored and collected. This enabled correlations between artificial traffic and actual network traffic to be calculated. The correlations provide a measure the accuracy of the traffic generator. A discussion of the correlation mechanism used in our artificial network traffic generator is given in Chapter 3.

### 1.4.4 Trafgen

Trafgen[13] is a program designed to artificially generate network traffic that can approximate the characteristics of real world traffic on the application level. Trafgen requires many steps to be taken before being able to generate the traffic. These steps can be summarized as follows:

- Data acquisition which includes monitoring the targeted network and collect the seed data
- Data interpretation, which includes grouping packets into their corresponding

connections, and including all the necessary information that can identify each connection (using *tcptrace*)

- Statistical processing which includes manipulating the data to create the cumulative probability distributions for the random variables defining the protocols and the other key parameters through *tcplib*.
- use the TCPLib generated information to generate an equivalent traffic, sent to *tcpdiscardD*

All the steps except the last use the tools previously discussed. The traffic generator engine is a multi-threaded program. This allows to model more accurately real connection, as it allows concurrent connections to co-exist on the network instead of the original sequential approach proposed by the original *tcplib*. The traffic generated by *trafgen* is monitored and later analyzed to study the correlation with the original traffic. It's important to notice that *trafgen* model the original traffic at the application level, not at the user level which leaves the scaling problem an unresolved one.

## 2. USER LEVEL MODEL

This chapter discusses the various implementations and changes made to different tools required to create an artificial traffic generator at the user level.

### 2.1 Design

The various components of our design can be summarized as follows:

1. Data Acquisition

Derive and interpret data from raw packets data.

2. Statistical Processing

Generate the breakdown functions representing the traffic

3. Traffic Generation

Generate the artificial traffic based on the breakdowns found in the previous step.

4. Data Verification

Correlate the artificial traffic with the real one.

The following sections describe each component.

#### 2.1.1 Data Acquisition

The first phase in our model is monitoring and collecting data about the site or the network we want to model. The most common way to obtain data about networks is to use network tracing programs such as *tcpdump*[19]. By using *tcpdump*, the data packets circulating over the targeted network are stored in trace files. These files

constitutes our raw data material. From these raw files we extract data in different forms suited to the statistical manipulation done in the second phase.

In our model, the Data Acquisition phase is separated from both the Statistical Manipulation and the Traffic Generation phases. This separation has the advantages that can be summarized as follows:

- The separation of the different phases into modules allows for the trace files to be preprocessed as an independent task from traffic generation, thus making the traffic generator quicker and smaller.
- It allows for changes in each phase to take place without the need to modify any of the other components as long as the right interfaces are maintained.
- It permits multiple raw files to be merged together.
- It permits users to manually change the data sets.

The main tool used for Data Acquisition is *tcptrace*[21]. The *tcptrace* engine was used to perform the necessary analysis on the raw data files. In addition to that, the fact that *tcptrace* is a modular program makes it easy to take advantage of the information *tcptrace* collects and write multiple purpose modules on top of that.

*Tcptrace* was used to interpret the raw data files and group them into connections and users. It performed four major functions:

1. Global Data Acquisition
2. Application Specific Data Acquisition
3. User Specific Data Acquisition
4. Data file Creation

### 2.1.1.1 Global Data Acquisition

Global Data Acquisition focuses on collecting information about the overall network traffic. In our case, the type of information we are interested in is:

- users inter-arrival times
- users frequencies

A user is defined to be all connections initiated by the same IP address over a certain period of time. If  $na$  is the number of applications, then the total number of partitions  $np$  of this set is given by:

$$np = 2^{na} \quad (2.1)$$

if we designated  $nu$  to be the number of users, and excluding the empty set cause it doesn't have a physical significance, then we have:

$$nu = 2^{na} - 1 \quad (2.2)$$

Given that we are only concerned about four applications *http*, *telnet*, *ftp*, *smtp* the number of distinct users is then 15.

The same IP address can represent more than one user. If the time between a new connection initiated from this IP and the most recent one before it bypasses certain threshold time value, then the IP address is considered a new user. This time threshold is determined as part of the configuration of the module extracting users from connections. User inter-arrival time is then the time used to determine when to start a new user of a certain type.

Users inter-arrival times are calculated by taking the difference between the start time of the first connection of the current user and the start time of the first connection of the previous user. The starting time of a user is always recorded in the data structure relative to the user.

The other type of global information is user frequencies. User frequencies are a measurement of how often users of a particular user type are seen on the network. Given this information we can then determine what type of users is going to be created next by the generator. The number of different types of users is known in advance and is determined by the number of applications studied.

#### 2.1.1.2 Application Specific Data Acquisition

The first type of interpreted data derived from the raw packets is connections. To model and extract connections from packets we need to break the Application Specific Data Acquisition down into two tasks:

1. Group the packets into their corresponding connections
2. Compute the parameters defining each connection

Knowing the source and destination ports of each packet makes it possible to infer which application that packet belongs to. Once the connections are identified, extracting useful information about each studied application is a straightforward job.

As was mentioned earlier, there are four types of applications modeled in our study. Each of these applications has a certain number of parameters that define it completely. These parameters are defined in [6] and [13]. For example to model a *telnet* connection, the following parameters are needed:

- Packet Size- the size of the data packets.
- Duration- the total duration of the *telnet* connection.
- Inter-arrival time- the time separating two consecutive *telnet* packets.

*tcptrace* is already equipped with a module to extract the application specific data from raw files. Since our model introduced the concept of users, and in order to take advantage of the existing tool, we needed to break down the original data files into separate raw files, with each one containing applications representing one type of users, and then run the existing module over them.

### 2.1.1.3 User Specific Data Acquisition

As mentioned earlier, our model differs from the previous model discussed in [13] by introducing the concept of users. For each user type, we have to collect information about the different applications forming such user. The parameters needed are:

1. Applications inter-arrival times.
2. Applications frequencies
3. User's duration.

In order to fully model a user, we need to know when this user starts a new application, and then what is the type of this application. Once determined, the application specific data can be obtained as described in the section 2.1.1.2. Also needed is the user's life. We need to know how long this user will keep initiating connections, which is determined by taking the difference between the time of the first packet of the first connection of this user with the time of the last packet of the last connection.

As in the case of Global Data Acquisition, User Data Acquisition first bundles all the connections belonging to one user together. To do this, we put together all the connections initiated by the same IP address with an application inter-arrival time less than the configured time threshold. We also maintain a data structure to record, among other things, the starting time of the user, and a pointer to a list containing the inter-arrival times of its applications. Once we have finished processing the data files, we then process users generating the three breakdowns mentioned earlier. This is done through a new plug-in module built on top of *tcptrace*.

### 2.1.1.4 Data File Creation

The last major function of the Data Acquisition Phase is to put the information generated in a format suitable for statistical processing. Since the next phase is the statistical processing phase and is done by *tcplib*, the generated data should conform

to the format expected by *tcplib*. To accomplish this task, we transform the extracted information into a cumulative probability function and then we store it in files in the form of histograms.

Two modules were used to achieve this last function. One is the newly built *user* module that generates the necessary files for the user specific data, and the other module is the existing *tcplib* module of *tcptrace* for the application specific data. The application specific data is only meaningful in the user context. Therefore before invoking the *tcplib* module, we break down the original data files into separate files each corresponding to an existing user type. Once finished the histograms are generated and stored separately for each existing user type.

Merging different files is also a point worth mentioning. By merging different packets from different data sets taken from the same network, we can mitigate the effect of possible anomalous behavior observed on that network. If during a monitoring test the network exhibited an anomalous behavior which is not observed later, the addition of more data sets taken from the same network reduces the effects of this anomaly by diluting this effect into a larger amount of data. *tcptrace* is able to process more than one file sequentially, one file at a time, and the modules maintain their information in a preprocessing mode till the end of the last file is reached.

### 2.1.2 Statistical Processing

The core of the Statistical Processing is *tcplib*. As mentioned in chapter 1, *tcplib* is a network traffic simulation library. Information in the form of cumulative probability distribution is fed to *tcplib* which in turns produces the necessary information for a conversation level emulator to use.

#### 2.1.2.1 *tcplib* Functionality

Originally *tcplib* accomplished two basic statistical operations:

1. Obtaining the application associated with the next connection using the applications' breakdown probabilities.

2. Determining, through a histogram map lookup, the characteristics of the application chosen in the previous step.

$$P_r(x) = \frac{x^{r-1}e^{-x}}{\Gamma(r)} \quad (2.3)$$

$$\sigma_{Erlang} = \frac{1}{\sqrt{r}}\left(\frac{1}{\eta}\right) \quad (2.4)$$

$$\Gamma(z) = \int_0^{\infty} x^{z-1}e^{-x}dx \quad (2.5)$$

To accomplish the first step, *tcplib* uses the applications' breakdown to compute the mean, mean square, and the variance of the breakdown, and then applies a gamma distribution function to determine the application that should start next. A gamma distribution has a low enough coefficient of variation to produce random numbers with variance matching the given measurements; For this reason *tcplib* uses the Ahrens method for generating random numbers distributed as a gamma distribution of integer order  $r$ , the  $r$ -stage Erlangian distribution. Equations 2.3 and 2.4 gives the gamma density function and the standard deviation of Erlangian distribution. The gamma function is given in Equation 2.5 [6].

The second step is performed through the usage of the inverse transformation method. This method is fully described in Appendix 1. The basic idea is to extrapolate a uniformly distributed sequence of random numbers between 0 and 1 into a sequence of random numbers between 0 and 1 obeying a non-uniform probability distribution. Using this idea *tcplib* generates a sequence of uniformly distributed random numbers, then extrapolates them into the empirical distribution generated from the real network, and based on the new extrapolated sequence, *tcplib* determines its next choice.

This method is used for all calculations except for determining the identity of the next application.

### 2.1.2.2 *tcplib* Enhancement

Since users are not supported by *tcplib* and since the introduction of this concept changes the way information about applications is organized, we had to modify *tcplib*. The fact that *tcplib* was designed to be modular helped in the case where the improvement doesn't go beyond adding a new application, in this case it's enough to create a new configuration file with the characteristics of the new application and *tcplib* will take care of the rest. But in our case, more fundamental modifications were needed because we wanted *tcplib* to be user-aware, which required rewriting parts of the engine of *tcplib* itself.

The way *tcplib* works consists of taking breakdown files and generating a set of *.c* and *.h* files. Since in our situation we have users and for every user we have a set of applications' breakdowns associated with it, we modified *tcplib* to identify whether the breakdowns read are users' or applications' breakdowns, and then for each user the corresponding applications are processed, and their data structures are created in the generated *C* files. Some extra data structures were needed to accommodate the explosive expansion of the original ones, and to bundle them in users groups. We also added a *next\_user* function to play a similar role to the *next\_app* one in choosing the next user to start. Also, the automatically-generated *C* language source files and *.h* files were generated to load all the data structures related to the set of applications for each user.

changes to *tcplib* took two forms:

1. modifications to existing files.
2. additions of new files.

These changes can be summarized as follows:

For the existing files:

breakdown

modified to process users' and applications' breakdowns and to generate the appropriate data structures in a *.h* file.

#### brkdn\_dist

we added a function to determine the next user, based on the same gamma distribution.

#### brkdn\_dist

the corresponding *.h* file to the above *.c*.

#### tcpapps

modified to contain the prototypes of the new functions for next user, and user's inter-arrival times, as well as users' lives.

#### tcplibgenc

modified to generate the *C* files needed with user support. It basically modifies the generated functions to take one more parameter which is the user type and load all the users tables to accelerate the map lookup during the traffic generation.

#### tcplibgenh

also modified to generate the *.h* files corresponding to the generated *C* files described above.

#### *.tc* files

these files are the *tcplib* configuration files. They are modified to hold the parameters of each application for all users.

all the automatically generated *.c* and *.h* files

They're modified through the two files *tcplibgenc.c* and *tcplibgenh.c*.

for the new files added we have:

### user\_inter

Added files to provide functions for inter-arrival times of users. It consists of three files, the *.c* file for the implementation of the library function returning the user-inter-arrival time, the *.h* its corresponding header file, and the *.tc* configuration file.

users\_brkdn the *.h* file used to hold the data for users breakdowns.

### ulife

Added files to provide functions for users' lives. It consists of three files, the *.c* file for the implementation of the library function returning the user duration time, the *.h* its corresponding header file, and the *.tc* configuration file.

The modified *tcplib* used the same statistical principles as the original but with relatively different data structures.

## 2.1.3 Traffic Generation

Once data is statistically processed and the network simulator library is ready to be invoked, the natural step is to start the traffic generation. The traffic generator relies on the information provided to it by the *tcplib* functions and generates its traffic accordingly. A program for traffic generation is already in use and was developed at Ohio University [13]. Given the separation between the traffic generation phase and the previous steps, it is possible to not only model a real existing network, but to also create hypothetical situations where certain information can be defined based on the imagined scenarios. The description of the old traffic generator and the enhancements done to it are the topics of the next sections.

### 2.1.3.1 Original Design of *trafgen*

*Trafgen* is a multi-threaded program that uses *tcplib* functions to generate artificial traffic resembling real network traffic between two machines.

To reflect the dynamic with which traffic is created, *trafgen* used multiple threads

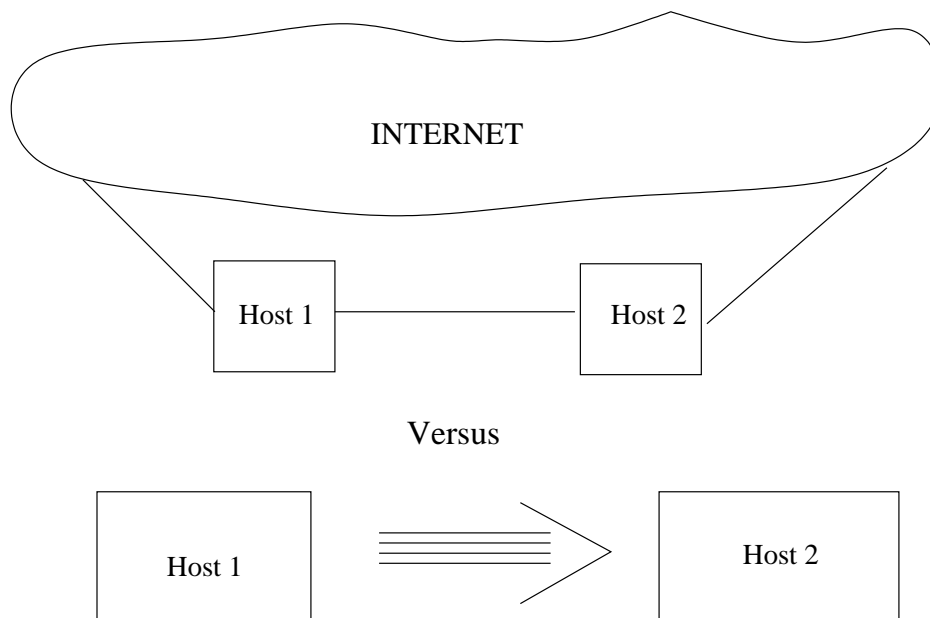


Figure 2.1. The TrafGen Model. This is how trafgen models the Internet. The internet between the two hosts is compressed through the first phases into a direct communication between two machines only, one client and one server.

to emulate the way real world connections are initiated. In real networks environments one or more hosts generate the traffic, and on each of these hosts one or more process is responsible for this hosts' traffic. In the original design of *trafgen* all the traffic coming from all the hosts is considered competing for the network resources the same way, in other words it's similar to a one big host generating competing traffic. A multi-threaded approach would be an optimal approach for the emulation of such environments. The way these threads are combined mirror more or less the accurately the networks studied. Originally *trafgen* used two types of threads:

1. "tcplib" threads.
2. applications' threads.

*trafgen* starts by creating a main thread called the "tcplib" thread, this thread is

responsible for the following:

- determine which application to start next
- start a new thread to handle this application
- sleep for the necessary interarrival time

The “tcplib” thread doesn’t generate any traffic, it simply interacts with the *tcplib* functions, spawn applications’ threads, and act as a scheduler for these applications’ threads. Through its interaction with the *tcplib* functions the “tcplib” thread determines the type of application to start next, and the time when that application should be started. Once known, “tcplib” thread spawns an application thread to handle the characteristics of the application as well as starting the connection itself. After the creation of an application thread, “tcplib” consults *tcplib* to find the time for the next application and sleeps waiting for the corresponding amount of time to elapse. The initialization phase refers to initializing main variables, such as the destination host, port offset, duration. a schematic representation of these steps is given in figure 2.2.

The applications’ threads are the ones responsible for artificial traffic generation. The application thread gets the necessary information from the “tcplib” thread upon creation, this information includes host name, the application type, port offset. Once created, the application thread determines the characteristics of the application associated with it through *tcplib* functions. Depending on the type of application, its corresponding thread will behave differently. The behavior of the different application threads is described as follows:

- *telnet* thread: *telnet* applications are the most unpredictable applications, and this is caused mainly because of the interactive nature of *telnet*. *telnet* threads start by initializing their destination host and port number inherited from the “tcplib” thread. Next the thread determines the duration of the *telnet* connection. Once determined the *telnet* thread starts sending packets, the packet’s

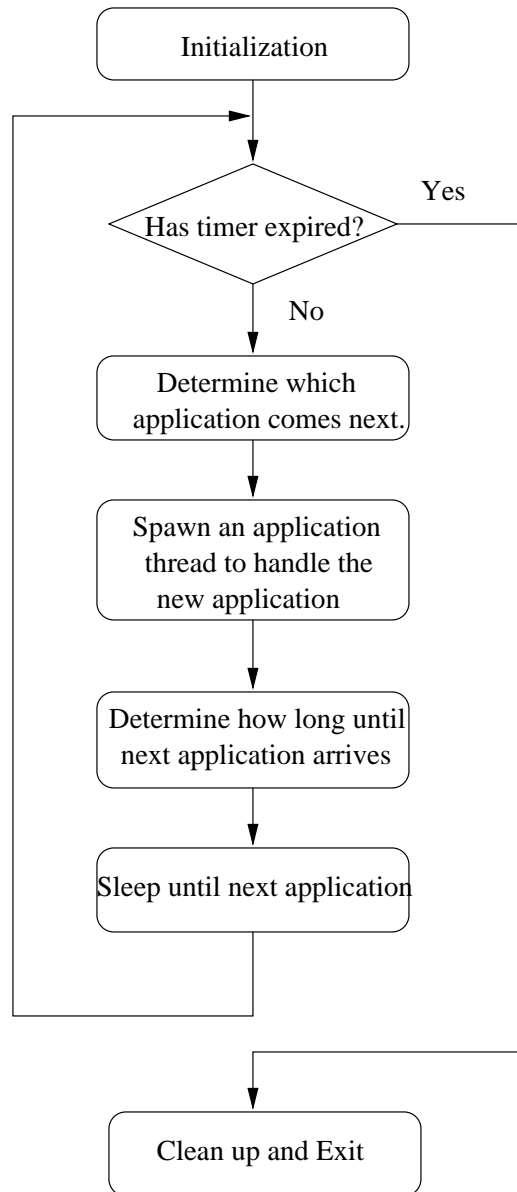


Figure 2.2. TCPLib Thread Flowchart. This flowchart describes the execution path of TCPLib threads.

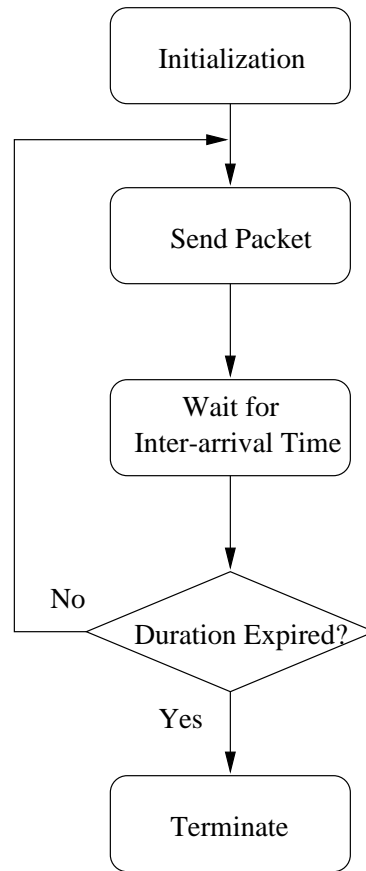


Figure 2.3. TELNET Application Thread Flowchart. This flowchart describes the execution path of TELNET application threads.

sizes are also determined through *tcplib*. It then waits for an amount of time equal to the interarrival time returned by *tcplib* before sending the next packet. This process continues until the duration expires. figure 2.3 shows a schematic representation of the *telnet* thread.

- *ftp* thread: as in the case of a *telnet* thread, the initialization phase consists of inheriting the destination and port number from the parent thread. The *ftp* thread then determines the number of data connections it has to generate, as well as the control sizes. Once determined, *ftp* data connections get serviced

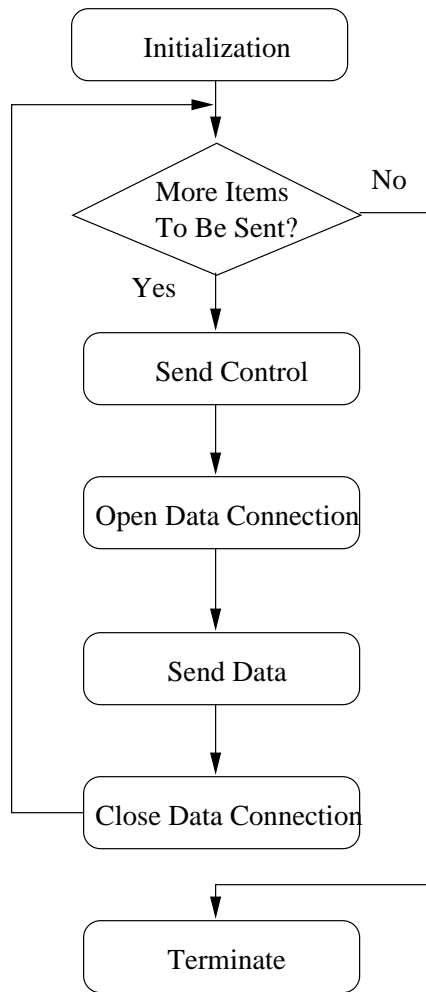


Figure 2.4. FTP Application Thread Flowchart. This flowchart describes the execution path of FTP application threads.

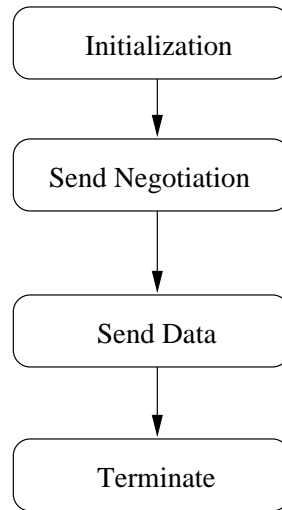


Figure 2.5. HTTP Application Thread Flowchart. This flowchart describes the execution path of HTTP application threads.

one at a time. The transfer size is determined by *tcplib*, and the data connection is then established, then the data is sent according to the information already collected. Figure 2.4 shows a flow chart of the *ftp* thread.

- *http* thread: following the common initialization phase, The connection between the two machines is established and the client is ready to start a negotiation with the server to send data. The negotiation between the two hosts consists of a set of questions and answers defined by the protocol specifications. Once the negotiation is over, the thread determines the transfer size through *tcplib*, the application thread then initiates the transfer, finishes and closes the connection and the thread consequently terminates. Figure 2.5 shows a schematic representation of the *http* thread.
- *smtplib* thread: the *smtplib* thread, after the initialization phase, determines the smtp item size from *tcplib*. Then a connection is established and a phase of negotiation is entered, once the negotiation is complete, the application thread

sends the data using the *smtp* item size previously determined. The end of the transfer terminates the connection and the thread itself. Figure 2.6 shows the *smtp* thread flow chart.

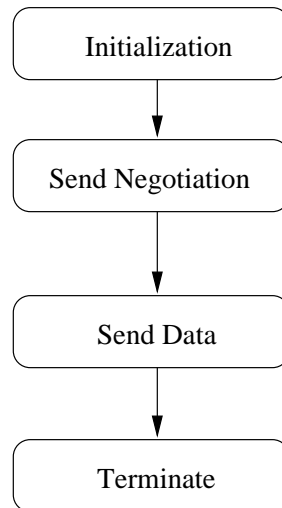


Figure 2.6. SMTP Application Thread Flowchart. This flowchart describes the execution path of SMTP application threads.

### 2.1.3.2 Enhancements to *trafgen*

As seen earlier *trafgen* is designed at the application level, this implies that *trafgen* means to add traffic to a network is by adding more connections. the situation resemble one big host generating all the applications. such solution doesn't seem to be right in the case of traffic scalability. Observations are showing that doubling the number of connections doesn't double the traffic generated. Our proposed solution to this problem is to create similar layers in *trafgen* as in the real world, in other words make *trafgen* user-aware. The introduction of users introduce one more layer of indirection in the thread hierarchie. Our new *trafgen* have the following threads in the same hierarchial order:

- “tcplib” thread as the main thread.

- users threads representing one of the 15 types of users.
- applications' threads similar to the previously described applications' threads.

In our new scheme, the “tcplib” thread, instead of creating an application thread, creates a user thread using the enhanced *tcplib* library which is now equipped by a function to determine the next user based on the users breakdown discussed earlier. Once created, the user's thread, based on its type, determines its own life time, and then starts using *tcplib* to decide on the next application to start and when, in other words the user thread plays a role similar to the “tcplib” thread in the original design. The application thread once created behaves like the old application thread in the original *trafgen*. Figure 2.7 shows the new design flowchart.

The following is a description of the new threads scheme:

- “tcplib” thread: It plays a similar role to the original one. It acts as a scheduler and a thread spwaner. First it determines what user to start next, and when. Once the user's identity is identified, the “tcplib” thread spwans a user thread to handle the new user. “tcplib” then determines the interarrival time before a new user is created and sleeps for that amount of time.
- user threads: users threads get their type from the “tcplib” thread. They are responsible for, first, determining the user's life time. This is done by a call to a new *tcplib* routine that determines the user's life. Once their life is known, user's threads start to determine what application goes next. It's important to note that not any application is a valid application. The type of valid applications depends on the user type, which is itself determined through the user's breakdown, if a user is defined as one that only uses *telnet* and *ftp*, this user won't have other type of applications in its breakdown, so any application other than *telnet* and *ftp* will not be initiated by this user. After determining the type of application to start next, the user's thread starts an application

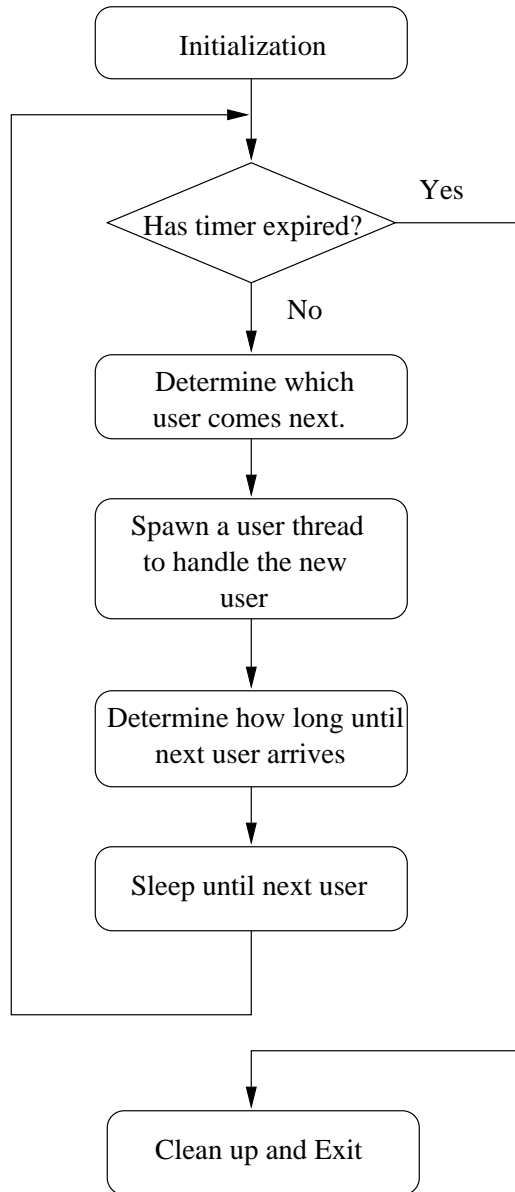


Figure 2.7. The User Model. This is how *trafgen* models the Internet from a user point of view.

thread to handle the application, and then it determines the interarrival time between its own applications and sleeps for that amount of time to wake up later and starts a new application. One important point here is that in this work we didn't study the correlation between different variables. One manifestation of this problem is situations where the life time of a user could be less than a *telnet* connection initiated by this user itself for example, in such situations we enforced the time of the user over the *telnet* one.

- application threads: The application threads are created by the user threads, instead of directly by the “*tcplib*” thread. They perform exactly the same way as in the original *trafgen*.

### 3. MODEL VERIFICATION

This chapter discusses the verification of our approach by addressing the issues involved in this verification, and the methodology used to perform it, showing how our model accomplished its goals in generating statistically accurate artificial traffic. The discussion includes the seed data used, the comparisons of several seed data sets, the experimental procedures, and the correlations between the emulated traffic and its data seed as well as a discussion of problems observed during and after the data generation.

#### 3.1 Seed Data

As discussed earlier, the purpose of our approach is to be able to create network traffic with the same characteristics as another existing network, and to identify the similar patterns in the two. In the previous chapter we discussed the design of our model, as well as the methods of transforming raw packets data into suitable data for statistical processing. This section discusses our raw data files, their source, the format of those files, and the methods used to collect this data.

To collect the raw data, we scheduled different tests over a one week period to collect data at different times of the day. These files formed our raw trace files. Each one of these trace files form a study data set. As described in chapter 2 these raw data files are passed through *tcptrace* and its different modules to generate the necessary files for *trafgen* to use. For the purpose of this work two networks were monitored and used as our trace files source. these two networks are:

1. New-Wave: A New-Wave Communications switched subnetwork.

The data was collected on [parrot.newwave.net](http://parrot.newwave.net).

## 2. OU Dorm: The Ohio University Dormitory Network.

The data was collected on `elb.penguinpowered.com` which is a *Linux* machine in the dorm.

These networks were monitored using *tcpdump* [19] for a predetermined period of time. The monitoring was done at intervals of two to three hours during different times of the day, over a week. One major constraint in deciding on the time length of the tests, was the disk space. Depending on the time of day and the day of the week, the load on the network could become very high, and consequently some files can be excessively large. Although these files can be compressed, moving these files around, as well as compressing and decompressing them took a significant amount of time.

Once collected, the trace files were moved to a central storage facility in a compressed format. The next step was to process these files through *tcptrace* to generate the files needed by *tcplib* and organize them in directory hierarchies where *tcplib* can find them. Once in place *tcplib* takes over and generates the necessary header files as well as the automatically generated *C* files. *trafgen* is then ready to generate the traffic. The process of preparing the configuration files for *tcplib* consumes a large amount of time because it requires generating the *tcplib* files for each user, archive them in the corresponding directory, then identify which users are present and which are not, and then generate the configuration files, and make them aware of non-present user types.

### 3.2 Evaluation Method

As mentioned earlier, the objective of this work, is to artificially generate traffic that accurately mirrors real world traffic. The main questions that arise in this situation are how to measure the accuracy of this model, what criteria to use, and which parameters to compare. Since the model uses random numbers to decide on the traffic patterns it will create and send, the experimenters can either reproduce the same experiment using the same data seed and the same seed number for the

traffic generator, or use a different seed number but maintain the same data seed and in this case the experiment is not an exact replication of the previous one, but should be statistically equivalent. This statistical equivalence is the topic of the next sub-sections.

Another important point in this study is the periods of time used to collect raw data. Given the two to three hours time periods, these intervals fail to show the self-similarity phenomenon investigated in [24] and [30]. These studies show that Internet traffic exhibits long-range dependency. This self-similar nature of the traffic requires a long term monitoring of networks that could spread over several days, this kind of monitoring was not within our capabilities.

The next sub-sections discuss the comparison methods we used to determine the accuracy of our approach.

### 3.2.1 Comparing Data Sets

Up to this point we don't have a well defined test that can determine the accuracy of our model. For this reason we resorted to well known statistical tests that could be adapted to our situation. Two tests presented themselves as possible candidates: the chi-square test, and the Kolmogorov-Smirnov test. Both of these tests check for the goodness of fit. These tests can be used to compare the breakdowns of our artificially generated traffic with the original breakdowns of the real traffic. Although the Kolmogorov-Smirnov doesn't require any grouping of observations, and any guidelines for choosing appropriate cell sizes as opposed to the Chi-square that does, we used the Chi-square test for the fact that this test is designed for large samples and discrete distributions, while the Kolmogorov-Smirnov test was designed for small samples and continuous distributions [15]. A slightly more elaborate test based on Chi-square test is given in [23].

### 3.2.1.1 Chi-square test

The chi-square test is general and can be used for any distribution. It can be used for testing random numbers as well as for testing random-variate generators. the main steps for this test are:

1. Partition the sample space  $S_X$  into the union of  $K$  disjoint intervals.
2. Compute the probability  $b_k$  that an outcome falls in the  $k$ th interval under the assumption that  $X$  has the postulated cumulative distribution function cdf. Then  $m_k = nb_k$  is the expected number of outcomes that fall in the  $k$ th interval in  $n$  repetitions of the experiment.
3. The Chi-square statistic is defined as the weighted difference between the observed number of outcomes,  $N_k$ , that fall in the  $k$ th interval, and the expected number  $m_k$ :

$$D^2 = \sum_{k=1}^K \frac{(N_k - m_k)^2}{m_k} \quad (3.1)$$

4. For an exact fit,  $D$  should be zero. However, due to randomness,  $D$  would be nonzero, so if the fit is good, then  $D^2$  will be small. Therefore the hypothesis is rejected if  $D^2$  is too large, that is, if  $D^2 \geq t_\alpha$ , where  $t_\alpha$  is a threshold determined by the significance level of the test.

The chi-square test is based on the fact that for large  $n$ , the random variable  $D^2$  has a probability density function pdf that is approximately a chi-square pdf with  $K - 1$  degrees of freedom. Thus the threshold  $t_\alpha$  can be computed by finding the point at which:

$$P[X \geq t_\alpha] = \alpha \quad (3.2)$$

where  $X$  is a chi-square random variable with  $K - 1$  degrees of freedom. The probability density function of a chi-square variable is given by:

$$f_X(x) = \frac{x^{(k-2)/2} e^{-x/2}}{2^{k/2} \Gamma(k/2)} \quad (3.3)$$

Applying this method with a given significance levels would give an indication about the goodness-of-fit of the distributions, thus allowing for a method of comparison of our traffic.

### 3.3 Experimental Procedure

After establishing a baseline for the accuracy measurement, we needed to conduct tests to determine the goodness-of-fit of the artificial traffic and its seed traffic. The testing procedure for traffic generation involved setting up *trafgen* to emulate a particular network, transmitting and monitoring the artificial network traffic generated by *trafgen*, generating *trafgen* data files based on the artificial traffic, and then apply the chi-square test to verify how good the data would fit.

The entire experimental procedure was designed to be composed of a series of two to three hour long tests. For all our tests, we used computers on the OU IRG network as the source and destination hosts. *pride*, a Sun Sparcstation 5, was used as the destination host where *tcpdiscardD* was running. The other machine was *uger* an Ultra-10 Sun workstation, and it was used as the traffic generator.

At a predetermined time, we initiated the tests by selecting a data set as a seed set. For each test, The data files in the seed set were used, *tcplib* will be recompiled with the new data files, and then *trafgen* gets recompiled itself to use the new data integrated into the newly compiled *tcplib*. On *pride*, the *tcpdiscardD* server is started, as well as *tcpdump* to monitor the network. *uger* then starts the traffic generation by launching *trafgen*. Once the traffic generation was complete, the *tcpdump* trace file was passed through *tcptrace* and the *tcplib* module to generate the necessary information extracted from the artificial traffic this time. The last step is to pass the generated breakdowns along with the original ones through the chi-square test and verify the accuracy the test had achieved.

### 3.4 Correlations of Artificial Traffic

The heart of the verification process for the artificial traffic is based on comparing the output of *trafgen* to the original network traffic that *trafgen*'s output was based on. Our goal has always been to create traffic that had the same characteristics as our seed traffic. Measurements of these relationships are difficult, even with the tools we have discussed in this Chapter.

The ideal output of *trafgen* would not be traffic exactly like what we had measured. This output can be generated using a simple packet level emulator. What *trafgen* attempts to provide is pseudo-random network traffic with the same general set of characteristics as a real network. It's true that computers generate the same sequence of random numbers given the same seed, but since *trafgen* is multi-threaded the choice of each thread that is going to choose a set of random numbers is totally based on random events, thus making the repetition of the same experiment with the same seed data producing a different traffic but preserving the same patterns. As described in this chapter, we used the chi-square method to determine how close the data we have mirror the seed data. Based on our results we made some conclusions about future enhancements to our method. The following tables show the results of one of the tests. For each test we present a comparison of user's percentages, the correlation between the lives of the users, and for each user type present in the test, we present the correlation of the different application's parameters. For each user type, we divided the sample space into intervals, then calculated  $D^2$  for each parameter, then obtained  $\alpha$  using the inverse chi-square function. The calculation of the inverse chi-square function is done through *MATLAB* by using the function *chi2inv*.

User types are given in Table C.1 in Appendix C. The intervals,  $D^2$  and  $\alpha$  are given in each table for each parameter. The following test was conducted for a time period of two hours, which is the same period of its seed data. The user population from the original traffic was 1060 users, the one from the artificial traffic was 717 users, and the number of connections for the original traffic is 14921 connections that

for the artificial traffic is 9710 connections. This information is summarized in table 3.2. The conventions used are described in table 3.1.

Table 3.1 conventions used for each user type

Conv	Conversation inter-arrival times
FCS	File Control Size
FTS	File Transfer Size
HBS	<i>http</i> Burst Size
NTS	<i>NNTP</i> Transfer Size
STS	<i>SMTP</i> Transfer Size
TD	<i>telnet</i> Duration
TIA	<i>telnet</i> Interarrival Time
TPS	<i>telnet</i> Packet Size

Table 3.2 summary information for the test presented

General parameters	Test time	Number of users	Number of connections
Original	2 hours	1060	14921
Artificial	2 hours	717	9710

Tables 3.3 and 3.4 show the user frequencies as observed for the original and artificial traffic. Some numbers in the artificial traffic show a considerable difference from those in the original one. We believe that this discrepancy is caused largely because of the disappearance of a number of pure *http* users from the whole population due to their unidirectional connections, thus causing the percentage of other applications to be higher in the reduced population. Section 3.5.3 discusses unidirectional *http*

connections in details. Also it can be observed that some users present in the original traffic are not in the artificial one, those users are of very low percentage in the original traffic and consequently of very low probability.

Table 3.3 comparison of users' frequencies between original and artificial traffic

User type	U1	U2	U4	U5	U6	U8
Original	0.050	0.013	0.635	0.002	0.017	0.221
Artificial	0.1	0.031	0.577	N/A	0.013	0.200

Table 3.4 comparison of users' frequencies between original and artificial traffic continued

User type	U10	U12	U13	U14	U15	Others
Original	0.001	0.048	0.002	0.007	0.001	N/A
Artificial	N/A	0.078	N/A	N/A	N/A	N/A

Table 3.5 shows the comparison between the original user lives and the artificial ones. We can observe that the two lives correlate well except for user types 4 and 8 which are *http* users only and *smtp* users only. The discrepancy in *http* lives can be attributed to the problems with the *http* connections discussed in section 3.5.3. The problem with *smtp* requires further investigation.

Tables 3.6 to 3.11 show a comparison of results for each user type observed during the test. While the application parameters correlate well for some users as in table 3.6 they don't show the same level of correlation for others. In table 3.7 we can see that FCS and FTS do not correlate to any significant level. This could be due to the relatively small number of intervals used to compute  $D^2$  and  $\alpha$ . In general most

Table 3.5 chi-square values for user lives.

values	Life 1	Life 2	Life 4	Life 6	Life 8	Life 12	Life others
Intervals	15	4	80	5	16	20	N/A
$D^2$	18.95	15.44	309	6.12	401	20.12	N/A
$\alpha$	5%	0.1%	-	10%	-	5%	N/A

of the parameters having small number of intervals for their sampling space suffer from a low correlation, and this is due to the nature of the chi-square test which is designed for a large number of intervals and samples. When we have a small number of intervals, any difference between the sample numbers in any interval would impact more significantly the overall result due to the absence of more intervals to dilute this difference on the other hand with a large number of intervals, a difference in sample numbers becomes more insignificant if over a large number of intervals the samples were close enough. Another observation is the low significance level of TIA in table 3.6 and that of Conv in table 3.11 which could be affected by the presence of competing traffic that could saturate the link during the experiment time causing a change in the inter-arrival time of *telnet* packets and applications due to the exponential back-off algorithm in Ethernet driver. This problem is addressed in section 3.5.1. Another side of the problem is related to the clock granularity under Solaris. Solaris clock is unable to give any significant timing below 20 ms which rounds up all the inter-arrival times under that number to 20 ms, thus creating the discrepancy. In tables 3.8 and 3.11 we observe a low correlation for the HBS, this could also be part of the unidirectional *http* connections discussed in section 3.5.3. the discrepancy in *smtp* sizes in table 3.11 should be further investigated.

The results of another test are included in Appendix C. As can be observed from the given results, some parameters do not correlate well, while some do to a

Table 3.6 chi-square values for user type 1.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	13	N/A	N/A	N/A	N/A	N/A	13	312	301
$D^2$	18.95	N/A	N/A	N/A	N/A	N/A	17.2	321.4	299.2
$\alpha$	5%	N/A	N/A	N/A	N/A	N/A	15%	10%	10%

Table 3.7 chi-square values for user type 2.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	11	7	12	N/A	N/A	N/A	N/A	N/A	N/A
$D^2$	17.33	20.11	21.42	N/A	N/A	N/A	N/A	N/A	N/A
$\alpha$	5%	-	-	N/A	N/A	N/A	N/A	N/A	N/A

Table 3.8 chi-square values for user type 4.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	400	N/A	N/A	100	N/A	N/A	N/A	N/A	N/A
$D^2$	200.72	N/A	N/A	98.12	N/A	N/A	N/A	N/A	N/A
$\alpha$	5%	N/A	N/A	15%	N/A	N/A	N/A	N/A	N/A

Table 3.9 chi-square values for user type 6.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	160	40	3	35	N/A	N/A	N/A	N/A	N/A
$D^2$	154.97	45.45	100.2	39.01	N/A	N/A	N/A	N/A	N/A
$\alpha$	5%	20%	-	25%	N/A	N/A	N/A	N/A	N/A

Table 3.10 chi-square values for user type 8.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	521	N/A	N/A	N/A	N/A	310	N/A	N/A	N/A
$D^2$	100.21	N/A	N/A	N/A	N/A	201.77	N/A	N/A	N/A
$\alpha$	1%	N/A	N/A	N/A	N/A	5%	N/A	N/A	N/A

Table 3.11 chi-square values for user type 12.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	499	N/A	N/A	89	N/A	77	N/A	N/A	N/A
$D^2$	200.04	N/A	N/A	100.1	N/A	80.13	N/A	N/A	N/A
$\alpha$	10%	N/A	N/A	15%	N/A	20%	N/A	N/A	N/A

certain extent. Some discrepancies exist in the users' frequencies. We address these discrepancies in section 3.5.3. Another observation is that with a large number of intervals and large sample space, the correlation is better which conforms to the chi-square test requirements for better results. The investigation of the problems causing the low correlation is given in the following section.

### 3.5 Difficulties in Generating and Correlating Artificial Traffic

During the artificial generation phase, we were faced with a set of problems that affected our results. Close examination showed some identifiable problems that we managed, to solve or propose solution for with the given time constraints. We analyzed the symptoms, identified the causes, and eliminated some of the problems. The following sections describe these problems.

#### 3.5.1 Isolated Network

We started by examining our testing environment. When we were conducting our artificial traffic generation tests, we had failed to consider that the IRG network had its own traffic patterns. We believe that the presence of additional traffic, other than the intended one, competing for the network resources affected the accuracy of our experiments. This extra traffic could be causing problems when *trafgen* sends large amount of data which, with the presence of the additional traffic, makes the link between the two machines overloaded. The presence of congestion on the local network can cause discrepancies in the inter-arrival times of applications and *telnet* packets when competing over the bandwidth of the Ethernet, due to the exponential back-off algorithm used by the Ethernet driver. The full impact of extra traffic is not fully understood and requires further investigation. To work around this problem two solutions were possible:

- create a small subnetwork comprised of only two computers
- run the experiments at times when the traffic in the IRG lab is very low.

Given the time constraints, the first solution, although more accurate, required a fair amount of time to set a new machine with an isolated network interface. The second solution seemed more appealing. We decided to launch our tests late at night, where the network utilization is very low.

### 3.5.2 *tcplib* Interpolation

Another problem observed in our experiments was the *tcplib* interpolation problem. In many instances, when the breakdown of the original data under examination showed peaks in the distribution, *tcplib* seemed to have spread this peak over a range of data points. In other words, the peaks in the original data sets were not transferred into the artificial data set.

When *tcplib* generates its histogram map, it uses the granularity of the data reported to it to fix its own granularity. If *tcplib* reads in more than one point with the same occurrence percentage, it lumps all data with the same percentage together. When doing histogram map lookups, *tcplib* will interpolate to equally choose any point in the range of values with the same percentage. *tcplib* chooses values in sparse regions with the probability assigned to the parameter studied at the top of the sparse region, leading to a large statistical spread. So long as a peak is surrounded by data points with different percentage values, then the peak will remain completely intact. If any points immediately neighboring the peak have the same percentage value as the peak, then the peak will be spread among the various points in the range. A zero value next to a peak will be assigned some of that peak's weight. This Observation has also been discussed in two previous works [13] [11]

### 3.5.3 unidirectional *http* connections

When the *tcpdiscardd* is busy under high load with hundreds of connections, it stops handling new connections initiated by *trafgen*. This is especially important in the case of *http*. When the server ignores *http* connections, we end up with *unidirectional http connections*. *Unidirectional http connections* are connections that have

Table 3.12 *tcplib* Interpolation Results. This table shows excerpts from the TELNET packet size data files for a seed data set, and its artificial traffic. Note the peak in the original data set at packet size 488, and lack of the peak in the artificial data set.

Packet Size	Percentage	Count	Percentage	Count
	Original Data		Artificial Data	
480	0.953	1	0.973	0
481	0.953	2	0.973	0
482	0.954	1	0.973	0
483	0.954	1	0.973	0
484	0.954	0	0.973	0
485	0.954	1	0.973	0
486	0.954	2	0.973	1
487	0.954	0	0.973	6
488	0.966	236	0.973	9
489	0.966	2	0.975	3
490	0.966	2	0.975	9
491	0.966	2	0.975	7
492	0.966	1	0.975	7
493	0.966	0	0.975	8
494	0.966	1	0.975	9
495	0.967	3	0.975	10
496	0.967	0	0.975	11

packets flowing in one direction but have no acknowledgments flowing in the other direction. On wide-area networks, it is common to see unidirectional *http* connections due to the different routes the acknowledgments may take and the overloaded routers that they encounter, resulting in the acknowledgments being lost and thus the connection appears to be unidirectional. On our experimental local network, it is less likely to see unidirectional *http* connections because all the data (including acknowledgments) flow on the same link. The fact that we did see those connections required closer examination. Close examination shows that when *trafgen* attempts an *http* connection to *tcpdiscardd* it sets a time out for it. When *tcpdiscardd* is overloaded it doesn't acknowledge the attempted *http* connections, in other words ignores them, causing those connections to be timed out by *trafgen*. The presence of the attempted connection in one direction and the lack of any acknowledgment in the other created the unidirectional *http* connections observed in our artificial traffic. The *tcplib* module of *tcptrace* ignores this type of connections and doesn't include it in the *http* breakdown. It does so because it is almost impossible to model unacknowledged *http* connections. On the other hand some users are pure *http* users, ignoring their *http* connections results in eliminating them as users completely from the user's breakdown, thus resulting in an inaccurate artificial data when the number of these users is significant enough, which is usually the case during long runs. This difficulty can be addressed by isolating the network where the tests are running, to reduce the amount of processing the destination machine itself is doing, and by increasing the capabilities of *tcpdiscardd* to handle more connections with higher load. Tests over a long period of time (i.e. days) will exhibit this problem in more severe ways, as the number of users eliminated will grow higher, especially that the highest percentage of user's type are the pure *http* users.

#### 3.5.4 User To Application Correlation

In real world environments, the life of a user is the sum of the durations of the applications this user launches. As mentioned earlier, our model still lack a correla-

tion method between different parameters, and most significantly, our model lack a correlation between the user's life and the duration of its applications. A manifestation of this problem is observed when a *telnet* application for example has a duration longer than the life of the user. In our model we enforced the life duration of the user over the duration of that of the application, thus changing the duration of these applications from what was chosen for them by *tcplib* to a different value. This kind of scenarios is causing some of the disturbances in the *telnet* duration distributions.

## 4. CONCLUSIONS AND FUTURE WORK

In this thesis, we have presented the underlying need for a model to artificially generate network traffic. This thesis has examined the potential uses for an artificial network traffic generator as a tool for researchers creating and testing new networking protocols. We have explored various options during the design process of the model, as well as capitalized on existing models through expansion and enhancement. We have also outlined a test method to verify the accuracy of a model through goodness-of-fit tests. What follows in this chapter is a discussion of the conclusions we can draw from this effort, and areas of future work that could be explored.

### 4.1 Conclusions

The purpose of our model was to produce efficient, statistically accurate artificial network traffic that mirrors the expected user load observed on real networks. We believe that our model shows that an efficient, customizable, artificial network traffic emulator is feasible, and we believe that this model is a viable development of this idea. This model can be the first prototype in this direction.

The model that we developed was architected with close similarity to real world networks. In its implementation, we used multi-threaded design with two layers of indirection, for users and applications, allowing the model to function without wasting cpu time with busy wait polling. In terms of efficiency, we were able to create and service various loads of users ranging between 40 and 100 users at one time with over 450 simultaneous connections.

We have also shown that the artificial network traffic produced by this model was accurate in its relation to *tcplib*'s output data. Our tests show that the traffic

produced artificially is representative of the network it was modeling, both in terms of individual characteristics and overall patterns in many cases, as well as outlining some of the reasons of divergence of some data sets. By using *tcplib*'s implementation as a conversation level network simulator[6], we have provided accuracy with flexibility at a different level than the more traditional packet level models can provide. Through our model we were able to make some observations about the statistical methods now in use and their potential problems, opening the gate for further studies in the arena of network simulation and modeling.

## 4.2 Future Work

Through our work with the user model, we were able to identify several directions for future work:

### Other Transport Protocols

Recently over the Internet, there's a growing popularity of multimedia traffic, such as multimedia streaming for audio and video applications. This traffic is based on UDP. This UDP traffic is starting to form a very significant part of the overall Internet traffic. Taking this change in the traffic patterns, a key way to improve this model would be to add support for additional transport protocols other than TCP which is what our model support now. Also ICMP traffic would be another major protocol to study and include into the user model. It is important to notice that adding these protocols requires changes to the tools used to capture the traffic as well as the tools for data interpretation and extraction, all this in addition to the statistical processing and the traffic generation itself.

### Other Applications

Following the same argument stated previously, to increase accuracy more applications need to be added to our model. The model currently support the set of applications given in [6] with the addition of *http*. This, however, is not a

complete list, and does, in fact, omit certain applications that play an important role on the Internet. Some of the applications that fall into this category are *DNS* [20], *RPC* [27]. Adding support for these applications would certainly result in more accuracy modeling the traffic.

#### Inter-parameters correlations

In previous models as well as in ours, the relations between the different key parameters are ignored, and each parameter is modeled as an independent random variable. The correlations between for the different parameters of the same application are not investigated. For example ignoring such dependencies would result in situations where the life time of a user is shorter than the duration of one of its *telnet* connections. The investigation of this area requires changes to the core of *tcplib* and its statistical method.

#### Verification model

Although we had used the chi-square test as a method of accuracy testing, a more elaborate method is required to be developed and integrated in the model. In a more general sense, a fully functional statistical model needs to be developed for traffic evaluation, in terms of goodness-of-fit, discrepancy measurements, and data correlation. Some of these models are being introduced in different places [23] [10].

#### Modularization

The way the traffic generator is written makes it difficult to add new applications. To add another application, like DNS, would require a programmer to modify *trafgen*'s main program file. This arrangement would put *trafgen* itself at risk of being improperly modified leading to incorrect results. Using modules for each application would insulate *trafgen*'s main source file, and make it easier to add new applications. In fact, a scripting language could probably be created

that would allow users to define new applications quickly, easily, and without writing any C code.

#### Kernel Level Support

A major part of the behavior of certain network protocols is dictated by the kernel's implementation of these protocols. Different operating systems may have different time granularity, leading to a different behavior of the *sleep* function, or to time stamps not reflecting the real generation time for the packets leading to erroneous results. These types of problems are very kernel dependent, and a natural step would be to seek a kernel level support for the model's implementation.

#### Graphical User Interface

To make the traffic generation and the presentation of results more user friendly, a Graphical User Interface (GUI) would be a convenient development of the the implementation. This kind of GUI could also be web-based, making it possible to run remote experiments without losing the benefits of a GUI display.

## BIBLIOGRAPHY

## BIBLIOGRAPHY

- [1] ALLMAN, M. Improving TCP Performance Over Satellite Channels. Master's thesis, Ohio University, June 1997.
- [2] ALLMAN, M., HAYES, C., KRUSE, H., AND OSTERMANN, S. TCP Performance Over Satellite Links. In *Proceedings of the 5th International Conference on Telecommunication Systems* (Mar. 1997).
- [3] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., AND KATZ, R. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *ACM SIGCOMM* (Aug. 1996).
- [4] BALAKRISHNAN, H., SESHAN, S., AMIR, E., AND KATZ, R. Improving TCP/IP Performance Over Wireless Networks. In *ACM MOBICOM* (Nov. 1995), pp. 2–11.
- [5] COMER, D. E. *Internetworking with TCP/IP, Volume I, Principles, Protocols, and Architecture*, 3rd ed. Prentice Hall, 1995.
- [6] DANZIG, P., AND JAMIN, S. tcplib: A Library of TCP/IP Traffic Characteristics. Tech. Rep. CS-SYS-91-01, University of Southern California, Oct. 1991.
- [7] DEGERMARK, M., ENGAN, M., NORDGREN, B., AND PINK, S. Low-Loss TCP/IP Header Compression for Wireless Networks. In *ACM MobiCom* (Nov. 1996).
- [8] DURST, R., MILLER, G., AND TRAVIS, E. TCP Extensions for Space Communications. In *ACM MobiCom* (Nov. 1996), pp. 15–26.
- [9] FRAZER, K. D. NSFNET: A Partnership for High-Speed Networking, Final Report 1987-1995. Tech. rep., Merit Network, Inc., 1995.
- [10] FUCHS, E., AND JACKSON, P. E. Estimates of distributions of random variables for certain computer communications traffic models. *Commun ACM* 13, 12 (December 1970).

- [11] HAYES, C. Analyzing the Performance of New TCP Extensions Over Satellite Links. Master's thesis, Ohio University, Aug. 1997.
- [12] HELVEY, E. TRAFGEN: An Efficient Approach to Statistically Accurate Artificial Network Traffic. Master's thesis, Ohio University, Aug. 1998.
- [13] JACOBSON, V., AND BRADEN, R. TCP Extensions for Long-Delay Paths, Oct. 1988. RFC 1072.
- [14] JAIN, R. *The art of computer systems Performance Analysis*, 1st ed. John Wiley and Sons, INC, 1991.
- [15] KRUSE, H. Performance of Common Data Communications Protocols Over Long Delay Links: An Experimental Examination. In *3rd International Conference on Telecommunication Systems Modeling and Design* (1995).
- [16] KRUSE, H., ALLMAN, M., GRINER, J., OSTERMAN, S., AND HELVEY, E. Satellite Network Performance Measurements Using Simulated Multi-User Internet Traffic. In *Proceedings of the Seventh International Conference on Telecommunication Systems* (1999).
- [17] KRUSE, H., ALLMAN, M., GRINER, J., AND TRAN, D. HTTP Page Transfer Rates Over Geo-Stationary Satellite Links. In *6th International Conference on Telecommunication Systems* (1998).
- [18] MAH, B. A. An Empirical Model of HTTP Network Traffic. In *IEEE INFOCOM* (1997).
- [19] MCCANNE, S., AND JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX Conference* (Jan. 1993).
- [20] MOCKAPETRIS, P. Domain Names - Concepts and Facilities, Nov. 1987. RFC 1034.
- [21] OSTERMANN, S. tcptrace: TCP dump file analysis tool, Jan. 1996. <http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html>.
- [22] OSTERMANN, S. tcpdiscardd: TCP Discard Server, Sept. 1997. <http://jarok.cs.ohiou.edu/software/tcpdiscard/tcpdiscard.html>.
- [23] PAXON, V. Empirically Derived Analytic Models of Wide-Area TCP Connections. *IEEE/ACM Transactions on Networking* 2, 4 (August 1994).
- [24] PAXSON, V., AND FLOYD, S. Wide-Area Traffic: The Failure of Poisson Modeling, june 1995.

- [25] POSTEL, J. Transmission Control Protocol, Sept. 1981. RFC 793.
- [26] POSTEL, J., AND REYNOLDS, J. TELNET Protocol Specification, May 1983. RFC 854.
- [27] SRINIVASAN, R. RPC: Remote Procedure Call Protocol Specification Version 2, Aug. 1995. RFC 1831.
- [28] STEVENS, W. R. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, Jan. 1997. RFC 2001.
- [29] THOMPSON, K., MILLER, G., AND WILDER, R. Wide-Area Internet Traffic Patterns and Characteristics (Extended Version). *IEEE Network* 11, 6 (November/December 1997).
- [30] W. E. LELAND, M.S. TAQQU, W. W., AND WILSON, D. V. On the Self-Similar Nature of Ethernet Traffic. *IEEE/ACM Transactions on Networking* 2, 1 (February 1994).

## APPENDIX

## A. INVERSE TRANSFORMATION

### A.1 Principle

The method of inverse transformation is based on the observation that given any random variable  $x$  with a cdf  $F(x)$ , the variable  $u = F(x)$  is uniformly distributed between 0 and 1. Therefore,  $x$  can be obtained by generating uniform random numbers and computing  $x = F^{-1}(u)$ .

#### A.1.1 Proof

Given the distribution of a random variable  $x$ , it is possible to find the distribution of any nondecreasing function  $g(x)$  of  $x$  as follows:

Let  $y = g(x)$  so that  $x = g^{-1}(y)$ . Where  $g^{-1}$  is the inverse function of  $g$ :

$$F_Y(y) = P(Y \leq y) = P(Y \leq g(x)) \quad (\text{A.1})$$

for an  $X$  such that:  $Y = g(X)$  equation A.1 becomes:

$$F_Y(y) = P(X \leq g^{-1}(y)) = F_X(g^{-1}(y)) \quad (\text{A.2})$$

Now we select  $g()$  such that  $g(x) = F(x)$ , or  $y = F(x)$ , so that  $y$  is a random variable between 0 and 1 and its distribution is given by:

$$F(y) = F(F^{-1}(y)) = y \quad (\text{A.3})$$

and

$$f(y) = dF/dy = dy/dy = 1 \quad (\text{A.4})$$

The two equations A.3 and A.4 shows that  $y$  is uniformly distributed between 0 and 1.

The preceding observation allows us to generate random variables with distributions for which  $F^{-1}$  can be determined either analytically or empirically. First generating a sequence of random numbers uniformly distributed between 0 and 1, and then use  $F^{-1}$  to turn that sequence into a sequence distributed according to a given distribution.

### A.1.2 Example

To generate exponential variates, we proceed as follows:

The pdf  $f(X) = \lambda e^{-\lambda x}$

The CDF  $F(x) = 1 - e^{-\lambda x} = u$  or  $x = -\frac{1}{\lambda} \ln(1 - u)$

Thus, exponential variates  $x_i$  can be produced by generating a uniform variable  $u_i$  and by using the preceding equation to determine  $x_i$ . Since  $u$  is uniformly distributed between 0 and 1,  $1 - u$  is also uniformly distributed between 0 and 1. Therefore, the generation algorithm can be simplified to:

$$x = -\frac{1}{\lambda} \ln(u) \tag{A.5}$$

As a final note, it's important to observe that the CDF is continuous from the right, that is, at any point where there is a discontinuity, the value on the right of the discontinuity is used, thus, the inverse function is continuous from the left.

## B. EXAMPLE DATA FILE

This appendix contains an example of user data file as generated in the Data Acquisition phase by the analysis module of *tcptrace*. The first column contains users' lives in milli-seconds, the second column contains the cumulative percentage of users with lives given in the first column, the third column contains the cumulative number of users with lives in the first column, and the last column contains the number of users with the life given in the first column.

Users' Lives (ms) - total	% lives	Running Sum	Counts
0.000000	0.000000	0	0
1.000000	0.056604	3	3
551.000000	0.075472	4	1
591.000000	0.094340	5	1
631.000000	0.113208	6	1
651.000000	0.132075	7	1
991.000000	0.150943	8	1
1211.000000	0.169811	9	1
1341.000000	0.188679	10	1
1511.000000	0.207547	11	1
1691.000000	0.226415	12	1
1751.000000	0.245283	13	1
1811.000000	0.264151	14	1
1931.000000	0.283019	15	1

1941.000000	0.301887	16	1
1961.000000	0.320755	17	1
2001.000000	0.339623	18	1
2481.000000	0.358491	19	1
2951.000000	0.377358	20	1
2991.000000	0.396226	21	1
3061.000000	0.415094	22	1
3441.000000	0.433962	23	1
3991.000000	0.452830	24	1
4021.000000	0.471698	25	1
4281.000000	0.490566	26	1
9581.000000	0.509434	27	1
9801.000000	0.528302	28	1
11841.000000	0.547170	29	1
20951.000000	0.566038	30	1
23081.000000	0.584906	31	1
31851.000000	0.603774	32	1
75351.000000	0.622642	33	1
78491.000000	0.641509	34	1
81231.000000	0.660377	35	1
117451.000000	0.679245	36	1
179631.000000	0.698113	37	1
220901.000000	0.716981	38	1
256331.000000	0.735849	39	1
1411281.000000	0.754717	40	1
1470901.000000	0.773585	41	1
1473291.000000	0.792453	42	1
1511811.000000	0.811321	43	1

2292781.000000	0.830189	44	1
2315961.000000	0.849057	45	1
2526491.000000	0.867925	46	1
2790251.000000	0.886792	47	1
3303881.000000	0.905660	48	1
3881391.000000	0.924528	49	1
3983811.000000	0.943396	50	1
4239811.000000	0.962264	51	1
4822331.000000	0.981132	52	1
4837371.000000	1.000000	53	1

## C. CORRELATION DATA

This appendix presents the results we obtained during one test of the artificial traffic generation. The comparison was based as described in Chapter 3 on the chi-square test. For each data set, we present the value of  $D^2$  and the significance level  $\alpha$ . The method used involved computing the value of  $D^2$  and then use that value with the number of intervals used to compute the inverse chi-square function, this is done using *matlab*'s function *chi2inv*. Due to the volume of the data, and the complexity of computation we only present some of the data collected in our experiments. Users are numbered according to their types. The following list describes the meaning of every type number:

Table C.1 conventions used for each user type

Conv	Conversation interarrival times
user type 1	uses <i>telnet</i> only.
user type 2	uses <i>ftp</i> only.
user type 3	uses <i>ftp</i> and <i>telnet</i> only.
user type 4	uses <i>http</i> only.
user type 5	uses <i>telnet</i> and <i>http</i> only.
user type 6	uses <i>ftp</i> and <i>http</i> only.
user type 7	uses <i>telnet</i> , <i>ftp</i> and <i>http</i> only.
user type 8	uses <i>smtp</i> only.
user type 9	uses <i>telnet</i> and <i>smtp</i> only.
user type 10	uses <i>ftp</i> and <i>smtp</i> only.
user type 11	uses <i>telnet</i> , <i>ftp</i> and <i>smtp</i> only.
user type 12	uses <i>http</i> and <i>smtp</i> only.
user type 13	uses <i>telnet</i> , <i>http</i> and <i>smtp</i> only.
user type 14	uses <i>ftp</i> , <i>http</i> and <i>smtp</i> only.
user type 15	uses <i>telnet</i> , <i>ftp</i> , <i>http</i> and <i>smtp</i> .

Table C.2 comparison of users' percentages between original and artificial traffic

User type	U1	U2	U4	U5	U6	U8	Others
Original	0.0746	0.0606	0.6573	0.0023	0.0023	0.2028	N/A
Artificial	0.0700	0.0552	0.5899	N/A	N/A	0.2801	N/A

Table C.3 chi-square values for user lives.

values	Life 1	Life 2	Life 4	Life 5	Life 6	Life 8	Life others
Intervals	20	19	200	3	2	61	N/A
$D^2$	28.87	20.01	230.11	20.75	9.21	80.66	N/A
$\alpha$	5%	20%	5%	-	-	1%	N/A

Table C.4 chi-square values for user type 1.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	25	N/A	N/A	N/A	N/A	N/A	24	390	86
$D^2$	31.23	N/A	N/A	N/A	N/A	N/A	25.42	385.78	100.02
$\alpha$	5%	N/A	N/A	N/A	N/A	N/A	10%	10%	10%

Table C.5 chi-square values for user type 2.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	21	13	10	N/A	N/A	N/A	N/A	N/A	N/A
$D^2$	19.17	13.65	40.12	N/A	N/A	N/A	N/A	N/A	N/A
$\alpha$	5%	30%	-	N/A	N/A	N/A	N/A	N/A	N/A

Table C.6 chi-square values for user type 4.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	101	N/A	N/A	71	N/A	N/A	N/A	N/A	N/A
$D^2$	110.43	N/A	N/A	80.11	N/A	N/A	N/A	N/A	N/A
$\alpha$	10%	N/A	N/A	10%	N/A	N/A	N/A	N/A	N/A

Table C.7 chi-square values for user type 5.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	3	2	6	6	N/A	N/A	1	2	1
$D^2$	2.01	12.89	20.08	18.97	N/A	N/A	N/A	1.10	N/A
$\alpha$	35%	-	-	-	N/A	N/A	N/A	15%	N/A

Table C.8 chi-square values for user type 6.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	3	1	4	3	N/A	N/A	N/A	N/A	N/A
$D^2$	2.73	N/A	5.01	2.99	N/A	N/A	N/A	N/A	N/A
$\alpha$	15%	N/A	15%	20%	N/A	N/A	N/A	N/A	N/A

Table C.9 chi-square values for user type 8.

values	Conv	FCS	FTS	HBS	NTS	STS	TD	TIA	TPS
Intervals	99	N/A	N/A	N/A	N/A	52	N/A	N/A	N/A
$D^2$	106.53	N/A	N/A	N/A	N/A	50.99	N/A	N/A	N/A
$\alpha$	10%	N/A	N/A	N/A	N/A	20%	N/A	N/A	N/A

SAFA, ISSAM. M.S., November, 2000  
Electrical Engineering and Computer Science

A User Level Model for Artificial Internet Traffic Generation

Director of Thesis: Shawn D. Ostermann

The unprecedented growth of the internet and the rapid advances in computer networking technologies have raised important issues in the protocol design methodology. Today's protocol designers have an increasing need to test their protocols under realistic traffic load. While software simulation is a necessary step in the design process and can provide valuable information about the expected system characteristics, the existence of a tool that can go beyond the traditional simulation and is able to reflect closely real network traffic have become a great necessity. This need has led us to develop a new statistical model that takes into account the network traffic from the user level down to the application level. this thesis suggests a user level statistical model that utilizes traffic patterns that can closely mirror the expected user load without the necessity to have a real end user network for the test. Our model is a conversation level model and it uses a set of statistical and software tools to emulate the same type of traffic over the test network.

Approved: \_\_\_\_\_