Spring 2002

Self-similarity in network traffic

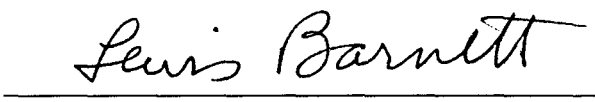
Francisco Chinchilla
University of Richmond

Self-similarity in Network Traffic

Francisco Chinchilla
Honors Thesis[1]
Department of Mathematics and Computer Science
University of Richmond
April 24, 2002

1. Under the direction of Dr. Lewis Barnett

This paper is part of the requirements for the honors program in computer science. The signatures below, by the advisor, a departmental reader, and a representative of the departmental honors committee, demonstrate that Francisco Chinchilla has met all the requirements needed to receive honors in computer science.



(advisor)

(reader)

(honors committee representative)

Self-Similarity in Network Traffic

Francisco Chinchilla
Mentor: Lewis Barnett
Department of Mathematics and Computer Science
University of Richmond, VA 23173

20th May 2002

Abstract

Recent studies have shown that various types of network traffic exhibit long-term dependency or self-similarity, or fractal-like behavior. Thus, to say that network traffic exhibits self-similarity means that if we look at the network traffic during a given time interval and choose a subinterval at random, the graph of the network traffic vs. time in the subinterval *looks like* the graph of network traffic vs. time in the original interval. This project will provide information on the pattern of the University of Richmond's network traffic and its self-similar properties.

1 Introduction

It is critical to properly understand the nature of network traffic in order to effectively design models describing network behavior. These models are usually used to simulate network traffic, which in turn are used to construct congestion control techniques, perform capacity planning studies, and/or evaluate the behavior of new protocols. Using the wrong models could lead to potentially serious problems such as delayed packet transmissions or an increase in packet drop rates.

Traditionally, packet arrivals were assumed to follow a Poisson arrival process. Although Poisson processes have several properties that make them easy to work with, they do not accurately describe certain traits seen in network traffic. Recent studies such as [1] and [2] have shown that LAN and WAN traffic exhibits a different kind of behavior than one would expect to see from

Poisson processes. One characteristic that differs is *burstiness*. A *burst*, or a period of intense activity[3], has no natural length in network traffic. If network traffic were to be a Poisson process, it would have a "characteristic burst length that would be smoothed by averaging over a long enough period of time" [1]. Instead, network traffic appears to be bursty in all time scales; it exhibits fractal-like behavior, a behavior described statistically by a self-similar process. A self-similar process by definition is one whose correlational structure remains unchanged regardless of the time scale being used. Thus, the burst lengths in a self-similar process will not be smoothed out. Instead, there will be bursty periods which themselves contain bursty periods, etc.

In Section 2 we discuss the related work that has been previously done on the self-similarity of network traffic. After discussing the process used to capture data in Section 3, we will statistically determine the fractal-like behavior in Section 4 and present estimates for this behavior. In Section 5 we will examine the results of our analyses, and then summarize and compare these results to those from previous studies in Section 6. Finally, acknowledgements are made in Section 7 and an Appendix containing more detailed information on the analysis performed (including source code and sample input and output for all the scripts used) is presented in Section 8.

2 Related Work

Previous studies such as [1] have analyzed network traffic and found it to be self-similar. The authors also found that although a linear increase in buffer size results in exponentially decreasing packet loss for Poisson traffic, applying this technique to a self-similar process causes the packet loss to decrease at a very slow rate. Although [1] studied network traffic in general, others studied certain types of traffic in greater detail as well.

For example, the authors in [2] focused on TELNET and FTP connections. They found that TELNET connections could be modeled as Poisson arrivals, but that the packet arrivals generated by these connections had a high degree of burstiness, and that the exponential arrival model that has been used greatly underestimated this. Similarly, FTP data connections were also found not to be faithfully modeled by a Poisson process. Finally, since interactive applications such as TELNET are usually favored by scheduling algorithms, applications such as FTP can be blocked for long periods of time.

In [4] a very in-depth analysis of HTTP traffic was done by modifying the web browsers. At the time, virtually everyone used the NCSA MOSAIC Web Browser to access the World Wide Web (WWW). The authors decided to modify the source code of MOSAIC so that it could provide them with extra information about each individual Web session. The authors then showed that while effective web browser caching, embedded images, sounds, and audio in html files did have an effect on self-similarity, the primary causes of self-similarity were *user think time* and *available file sizes*. User think time is just the time it takes a user to process the information the web page is displaying and make a decision as to where to click/go next. Available files has to do with the fact that multiple user requests for the same file may occur. They showed that the distribution of the these files' sizes have a greater impact than user requests.

We will examine (in addition to some of the previously discovered phenomena) the implications that the new Peer-To-Peer (P2P) file sharing programs have on network traffic and self-similarity versus ftp. Although many of these P2P programs are used in non-academic endevours and therefore have been (in some cases) restricted by University, local, and/or international laws, many researchers believe that using P2P software simplifies the collaboration process. The Packeteer PacketShaper software package [5] is installed in the University's network. Using this application traffic and bandwidth management system we learned that KaZaA [6]was (and still is) the most used P2P program. Packeteer also reported that KaZaA uses port 1214 for both inbound and outbound traffic. Therefore we analyzed all traffic on this port (we assumed that the number of people that change the default port number for an application is negligible) will give us insight into the properties of P2P traffic.

Also in recent years, we have been experiencing an increase in the amount of multimedia content that is transmitted across the internet. For example, videoconferencing, videophones, and live video broadcasts have all become increasingly popular due to the rise of high-speed internet connections. If we were really to have many virtual classrooms in the university in the future, where the class is really a video-conference, then we need to understand how video broadcasts work, what kind of compression techniques are used in the broadcast, etc. Most of the video today uses variable bit-rate compression technique, and the authors in [7] studied in detail the self-similarity in various different types of broadcasts. However, we did not find that this type of traffic is as predominant as some of the other types of traffic on our network. Furthermore, it is difficult to determine that the network connection is of this type simply by looking at

the port numbers of the sender/receiver. Thus, we decided not to analyze this type of traffic at this time.

The final type of traffic we looked at is America On Line Instant Messenger (AIM) [8] traffic. Even though AIM now has file-sharing capabilities, it is very popular in college campuses mainly for it's chatting capabilities. We do not expect AIM traffic to overload the server in terms of bytes transferred, but since it is very popular we do expect a substantial amount of packets being transferred. Furthermore, this type of traffic has user think time in between packet transmissions, and we would like to analyze what implications (if any) this has on self-similarity.

It must be noted that in [9] the authors saw the traffic on the internet backbone smoothing out, which (as noted in Section 1) is what you expect from a Poisson process. More specifically, they found that "as the active connection load on an internet link increases, the long-range dependence of network traffic begins to disappear and ... the long-range dependence of the inter-arrivals and sizes goes locally to independence". We do not expect this to be the case in our analysis, for our network usually operates only at about 15 Mbps of the maximum 45 Mbps possible.

3 Data Acquisition

3.1 Hardware

The University of Richmond's internet connection is a 45 Mbps DS3 link to Network Virginia that terminates on a Cisco 7206 router. All traffic passes from the router, through a Checkpoint firewall, to an Enterasys SSR8600 router. To monitor Internet traffic we mirrored the traffic onto a 100Mb Fast Ethernet port on *porky*, a Compaq Proliant DL320 server using RedHat Linux 7.2 with kernel 2.4.9-21 and libpcap-0.6.2-9. A schematic diagram is shown in Figure 1.

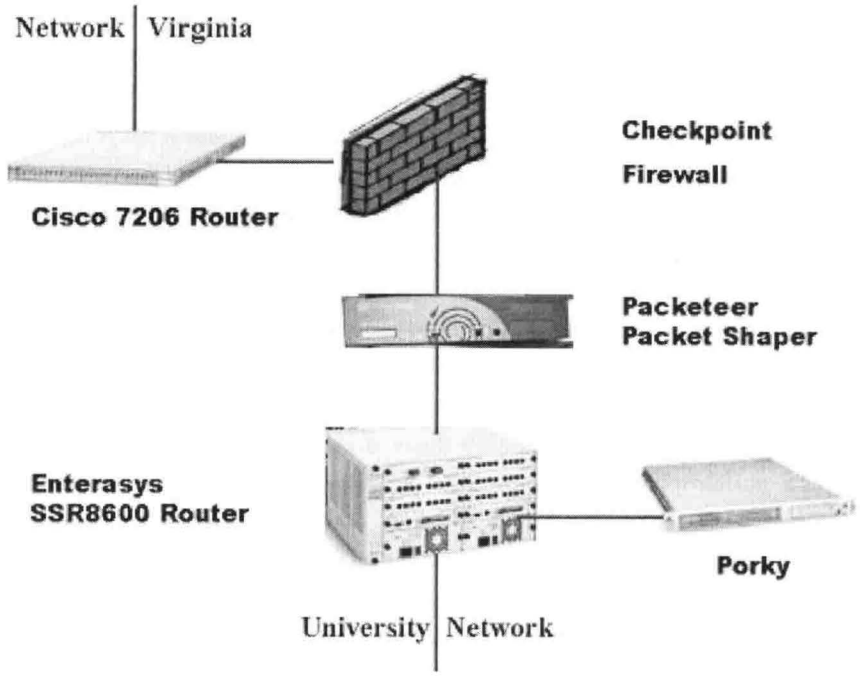


Figure 1: Schematic diagram of University of Richmond's network configuration

It must also be noted that Packeteer is used to limit certain types of outbound traffic, including P2P applications such as KaZaA, to a set number of Mbps. However, we were not able to study the impact that this limit has on self-similarity because this would have required reconfiguring the University's router structure. Packeteer calls setting this type of limit as "non-burstable traffic", and from what is explained in Section 4, we believe that this type of traffic would indeed have a lower degree of self-similarity.

3.2 Software

Much debate went into what piece of software should be used to capture network traffic. The analysis portion requires the timestamp of each packet arrival in order to analyze the network traffic. However, in order to be able to analyze different types of network traffic such as http, ftp, etc., the software used to capture the traffic data must also record the port that the packet was transmitted to/from. Furthermore, we also wanted to record the size of each packet. Since these kinds of demands are not uncommon, several software packages are able to record this type of information for each packet. At first we tried to

use the Open Source Intrusion Detection System, SNORT [10]. Snort had the advantage of being extremely flexible, but unfortunately all of the extra features made SNORT too complex to use as well as unstable; segmentation faults occurred after a few minutes of capturing. A more reasonable option seemed to be Ethereal[11]. Ethereal recorded exactly the pieces of information we were looking for, and it was able to output the information in various formats. Unfortunately, after approximately thousand packets, Ethereal proved to be unstable as well. We also tried the text version of Ethereal, tethereal, hoping that the lack of the GUI would help, but we were still getting an inconsistent response with segmentation faults occurring from time to time.

Instead of using one of these programs with *nice* output, we decided to use tcpdump[12]. This program is much simpler than the others, but in order to obtain the information needed, a lot more of extra information was also captured. As a result, our data captures were larger and scripts had to be developed in order to extract the pieces (mentioned above) necessary for our analysis.

We captured data for several days and then selected a set of files that would allow us to make the comparisons we wanted. The six time periods that we decided to present our results for are show below in Table 1. Unfortunately, due to our space limitations on the local hard drive, on average we could capture about 30 mins at one time, then had to stop capturing and spend 3 minutes to zip the file before we could continue.

Traces of Network Traffic

File		Start	End	Trace Duration	Packets	Bytes
Period	1	Thu Nov 29, 2001 4:17 pm	Thu Nov 29, 2001 6:55 pm	158 min	43,965,632	18,982,480,200
Period	2	Sun Dec 1, 2001 3:08 am	Sun Dec 1, 2001 5:43 am	155 min	15,972,722	9,902,661,608
Period	3	Mon Dec 3, 2001 8:48 pm	Mon Dec 3, 2001 10:41 pm	113 min	23,982,732	10,045,467,310
Period	4	Tue Dec 4, 2001 3:35 pm	Tue Dec 4, 2001 5:53 pm	138 min	31,974,774	17,019,364,843
Period	5	Tue Dec 4, 2001 7:33 pm	Tue Dec 4, 2001 9:23 pm	110 min	23,979,326	11,940,312,168
Period	6	Wed Dec 5, 2001 4:02 pm	Wed Dec 5, 2001 6:02 pm	120 min	27,978,756	14,124,925,478
Period	7	Thu Dec 6, 2001 3:35 am	Thu Dec 6, 2001 5:50 am	135 min	11,978,176	6,742,522,670
Period	8	Thu Mar 14, 2002 2:41 pm	Thu Mar 14, 2002 5:10 pm	149 min	37,548,395	16,941,178,120

Table 1: Details on each trace of network traffic

4 Mathematical Background

For a more in-depth discussion of the mathematics behind self-similarity and its estimates refer to [1] and [2]. The following discussion follows closely those in these two sources. However, this discussion tries to give only enough information to be able to understand how to compute the estimates for self-similarity. A more mathematically inclined reader is encouraged to read [13].

4.1 Self-Similarity

Intuitively, an arrival process that is self-similar is one whose arrivals exhibit fractal-like behavior. If we were to plot the number of packets vs. time unit on different time scales, the plots would be essentially the same. However, this intuitive description needs to be rigorously described mathematically. We begin by defining $X = \{X_t : t = 1, 2, 3, ...N\}$ to be the network packet arrival process, where each X_t is the number of packets that arrived in the *t-th* time unit. X is a covariant stationary stochastic process; that is, one with constant mean and finite variance. It's autocorrelation function ($k = \{1, 2, 3, ...\}$) is:

$$r(k) = \frac{E[(X_t - \mu)(X_{t+k} - \mu)]}{E[X_t - \mu)^2]}$$

Now, for each $m = \{1, 2, 3, ...N\}$, if we were to average each set of m non-overlapping arrivals, we would have an aggregated set, call it $X^{(m)}$ (for example, if $X = \{2, 4, 5, 9, 10, 10\}$, then $X^{(2)} = \{3, 7, 10\}$ and $X_1^{(2)} = 3, X_2^{(2)} = 7, X_3^{(2)} = 10$). Note that $X^{(m)}$ is also a covariance stationary stochastic process, and let $r^{(m)}(k)$ be the corresponding autocorrelation function for $X^{(m)}$. If X is self-similar, then the autocorrelation function $r^{(m)}(k)$ is the same for for all of the series $X^{(m)}$.

As a result, such processes exhibits *long-range dependence.* A process with long-range dependence has an autocorrelation function $r(k) \sim k^{-\beta}$ as $k \to \infty$, where $0 < \beta < 1$. Thus, instead of having an exponential decay that traditional traffic models display, self-similar processes decay follows a power law. Furthermore, since $\beta < 1$, the sum of autocorrelation values does not converge; it approaches infinity. As a result, the variances of $X^{(m)}$ do not decrease proportionally to $\frac{1}{N_m}$ as they do for uncorrelated datasets, but instead the variances decrease proportionally to $N_m^{-\beta}$ (here, N_m denotes the number of elements in the set $X^{(m)}$).

Finally, it must be noted that $X = \{X_t : t = 1, 2, 3, ...N\}$ could be any other type of arrival process, specifically a *byte arrival process.* That is, each X_t could represent the number of bytes received in the *t-th* time unit. Since most routers today employ congestion control techniques based on packet traffic instead of byte traffic, from here on we will be referring to packet traffic unless otherwise noted.

4.2 Estimates of Self-Similarity

Since slowly decaying variances and long-range dependence are both manifestations of the self-similarity in the covariance stationary stochastic process, there are several methods to estimate the degree of self-similarity. For practical considerations such as computational efficiency and step-by-step explanations on how to compute these estimates, see [14].

4.2.1 Variance/time

As noted in 4.1, the variances of $X^{(m)}$decrease proportionally to N_m. Let $\mu(m)$ be the mean of $X^{(m)}$ and $S(m)$ the standard deviation of $X^{(m)}$, and the normalized variance $\bar{S}(m) = S(m)/\mu(m)^2$. In a log-log plot of $\bar{S}(m)$ vs. m, the resulting graph is a line with slope $-\beta$, where $0 > -\beta > -1$. The closer $-\beta$ is to 0 (the flatter the line is), the higher the degree of self-similarity. Table 2 shows data taken from Period 8 and Figure 2 shows its corresponding Variance-Time plot.

M	Variance	Packets/10ms	Normalized Variance	Log(M)	Log(Normalized Variance)
1	349.3114	41.853533	0.199410715	0.00000	-0.70025151
5	232.2376	41.853598	0.132576636	0.69897	-0.877533006
10	208.9637	41.853667	0.119289915	1.00000	-0.923396271
25	182.3988	41.853745	0.104124584	1.39794	-0.982446719
100	156.0729	41.854276	0.089093867	2.00000	-1.050152193
125	153.3241	41.853745	0.087526942	2.09691	-1.057858247
625	140.6278	41.859254	0.080257969	2.79588	-1.095511835
1000	138.7352	41.856434	0.079188533	3.00000	-1.101337702
3125	132.8126	41.932938	0.075531607	3.49485	-1.121871276
10000	131.0149	42.022792	0.074190945	4.00000	-1.129649098

Table 2: Variance-Time analysis data for Period 8 packet traffic

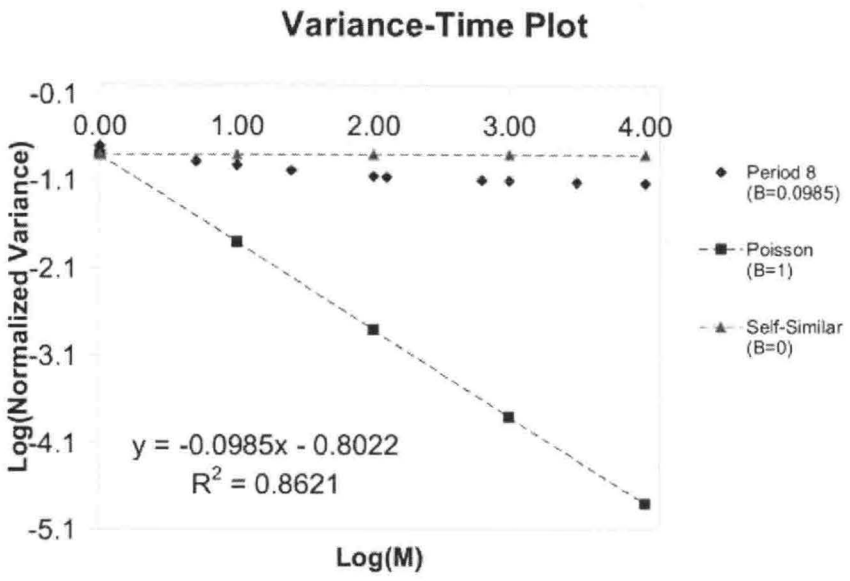


Figure 2: Variance-Time plot for Period 8 packet traffic compared to Poisson traffic and a self-similar process with $\beta = 0$

Our analysis shows that our estimate for β is 0.0985, or very close to 0 and thus has high degree of self-similarity. We now have strong evidence that our network traffic is self-similar, but our R^2 value (*coefficient of determination*, or how close our data points are to a straight line) for our least-squares fit is only about 0.85. We will now try a different method to estimate self-similarity.

4.2.2 R/S

In order to compute the degree of self-similarity using the R/S method, we first calculate the mean μ and the standard deviation S of X. Next, define each $W = \{W_k : k = 1, 2, 3, ...\}$, where

$$W_k = \sum X_k - k * \mu.$$

We then define R to be the maximum distance between any two W_k, that is, $R = MAX(W) - MIN(W)$.

We need to do this for several $X^{(m)}$'s, so define $W_k^{(m)} = \sum X_k^{(m)} - k * \mu(m)$ and $R(m) = MAX(W^{(m)}) - MIN(W^{(m)})$.

A log-log plot of $R(m)/S(m)$ vs. m for produces a line with slope H (the

Hurst parameter), where $0.5 < H < 1$. Here, the degree of self-similarity is higher when H is close to 1. Table 3 and Figure 3 are also from Period 8, only this time the R/S analysis is presented.

N	R/S Estimate	Log(N)	Log(R/S Estimate)
897138	251767.8153	5.9529	5.4010
179427	61754.2684	5.2539	4.7907
89713	32550.8511	4.9529	4.5126
35885	13936.1439	4.5549	4.1441
8971	3765.9129	3.9528	3.5759
7177	3039.9234	3.8559	3.4829
1435	634.4809	3.1569	2.8024
897	399.3332	2.9528	2.6013
143	64.7979	2.1553	1.8116
89	40.4143	1.9494	1.6065

Table 3: R/S analysis data for Period 8 packet traffic

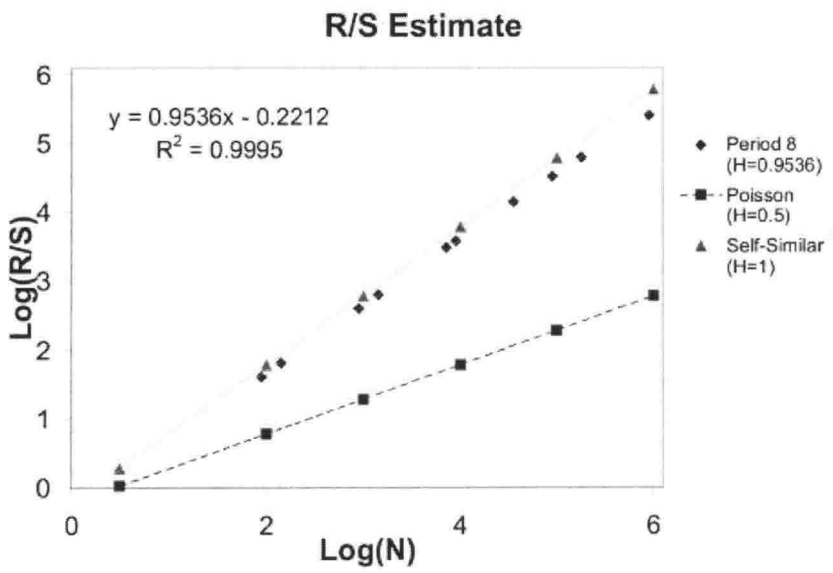


Figure 3: R/S analysis plot for Period 8 packet traffic compared to Poisson traffic and a self-similar process with $\beta = 0$

Our estimate for H is 0.9536, only that R^2 value is 0.9995, which is much higher. After repeating this procedure for various other traces I found that R/S analysis proved to have a higher value for R^2. We must ask ourselves which estimate is better, H or β? As it turns out, H is related to β via $H = 1 - \beta/2$.

Applying this to our previous estimate for β we get $H = 1 - 0.0985/2 = 0.9508$. Even though our estimate for β only had a coefficient of determination of about 0.85, it agrees to within less than 1% of our estimate for H. We found that they usually agreed to within 3% for other files, so our decision on which estimate to use came to a matter of computational efficiency and convenience.

One source [15]that not only described how they reproduced some of the results in [1], but also provided a link to some of the tools used [16] to calculate H using R/S analysis. Thus, the easiest estimate to compute was H, and since its R^2 value was higher than the one for β, we decided to use H. After computing several values for R/S we used a modified version of the least-squares fit software in [17] to compute the slope of our data points and determined H this way.

5 Results

We obtained estimates for H for each data collection period for each type of traffic (see 8.2). We then ran Analysis of Variance tests (ANOVAs) as described in [18] and observed the following:

5.1 Packets vs. Bytes

We found that for our sample, the estimate for H for packets was slightly higher than that for bytes. However, there was not (enough) evidence to reject the null hypothesis $H_{packets} = H_{bytes}$. That is, there wasn't evidence to support the claim that the degree of self-similarity of packet arrivals differs from the degree of self-similarity for byte arrivals. We found this to be true regardless of what type of traffic we were looking at.

5.2 Different Network Usage

In terms of packets per second, when the average number of $packets/s$ was high (Periods 1, 3, 4, 5, 6, and 8), our estimate for H was once again slightly higher than when the average number of $packets/s$ was low (Periods 2 and 7). Similarly, when the average number of $bytes/s$ was high (Periods 1, 4, 6, and 4) our estimate for H was slightly higher than when the average number of $bytes/s$ was low (Periods 2, 3, 5, and 7). Once again, there was not evidence to find that this is the case in general. Thus we conclude that we have no evidence that network usage has an effect on the degree of self-similarity in the University of

Richmond. We believe this to be due to the relatively low peak network usage.

5.3 Different Types of Traffic

We considered all of the network traffic, only HTTP traffic, only FTP traffic, only KaZaA traffic, and only AIM traffic, and found that their H estimates were not different for packet arrivals. That is, we did not find evidence that the degree of self-similarity of packet arrivals for each of these types of traffic differ. We did find evidence that for byte arrivals, network traffic overall and AIM traffic is more self-similar than KaZaA traffic (we cannot conclude anything about the degree of self-similarity of FTP and HTTP traffic in relation to the others).

5.4 Different Types of Traffic (With Network Usage Interaction)

Here we get to see what happens under certain specific conditions, without ignoring the interactions a condition could have. For example, when the average number of *bytes/s* was high, we found evidence that the degree of self-similarity of KaZaA byte traffic was lower than that of AIM byte traffic.

We also found that when the average number of *packets/s* was low, we found evidence that the network packet traffic was more self-similar than AIM packet traffic.

Finally, when the average number of *packets/s* was high, the *byte* traffic produced by AIM was more self-similar than the byte traffic produced by FTP traffic, and that KaZaA's byte traffic's degree of self-similarity was lower than all other 4 types of traffic.

6 Summary and Conclusions

Recent studies have shown that long-range dependence is exhibited by network traffic. This long-range dependence, or self-similarity, is not captured in traditional Poisson models used to model network traffic. Since congestion control mechanisms rely on and exploit certain properties in the model that is being used, using the wrong models could lead to problems such as delayed packet transmissions or even starvation.

We confirmed the existence of self-similarity in the University of Richmond's network traffic. The observed self-similarity was quite robust in all the different aspects that we considered. Even though in our sample data we found that packet traffic was slightly more self-similar that byte traffic, we were unable to find evidence to support the claim that the University of Richmond's packet arrival process is more self-similar than its corresponding byte arrival process. We observed that our sample data also had a higher degree of self-similarity when the network usage was high than when network usage was low. However, this difference was not statistically sufficient to infer that there is a difference in the self-similarity during periods of high network usage versus periods of low network usage in the University of Richmond's network traffic. Also, we found no evidence that (in general) there is a difference in self-similarity between the different types of traffic that we studied.

We did find evidence that, under certain circumstances, AIM traffic was more self-similar than KaZaA traffic. This leads us to speculate that a process with user think time has high degree of self-similarity, which supports one of the findings in [4]. Unfortunately we can say very little else. We were not able to directly detect possible causes of self-similarity (we did not have the ability to recompile the web browsers that everyone in the University uses or do something similar).

We can, however, say that since current congestion control techniques are more efficient at handling less self-similar traffic, if the entire University wanted to share files in a particular way, P2P would be an efficient choice (more efficient than AIM!).

7 Acknowledgments

The authors would like to thank the University of Richmond Network Services, in particular Holly Caruso, Greg Miller, and Coates Carter. Without their help and support we would have never been able to obtain and analyze the data needed.

We would also like to thank Van Bowen, Emeritus Professor of Mathematics at the University of Richmond, for spending some time discussing the statistical methods used.

8 Appendix

8.1 Trace Periods: Traffic Type-Specific Details

The following table contains the total number of packets and bytes for the different classes of network traffic we analyzed during each Period.

Trace Details									
		Period 1	Period 2	Period 3	Period 4	Period 5	Period 6	Period 7	Period 8
All Traffic	Packets	43,965,632	15,972,722	23,982,732	31,974,774	23,979,326	27,978,756	11,978,176	37,548,395
	Bytes	18,982,480,200	9,902,661,608	10,045,467,310	17,019,364,843	11,940,312,186	14,124,925,478	6,742,522,670	16,941,178,120
HTTP	Packets	17,140,393	2,738,469	10,212,694	13,469,747	9,963,995	12,347,176	1,998,769	12,574,900
	Bytes	7,029,963,138	1,085,037,545	4,318,810,878	6,385,768,745	4,543,461,264	6,034,076,026	705,733,193	4,977,843,622
FTP	Packets	1,489,350	616,008	705,801	715,785	472,136	1,530,165	1,088,683	60,733
	Bytes	1,241,841,316	511,300,253	572,031,517	620,841,812	359,039,672	1,252,131,271	1,008,431,186	9,098,321
KAZAA	Packets	7,466,917	3,902,754	5,081,170	3,295,443	2,694,830	3,182,218	2,167,857	2,989,171
	Bytes	3,153,854,421	2,974,556,534	2,283,897,244	2,202,973,931	1,823,899,821	2,238,887,661	1,653,092,627	1,827,662,267
AIM	Packets	2,907,813	739,855	2,328,383	2,871,682	2,570,281	2,065,911	644,745	2,484,522
	Bytes	782,200,367	78,280,429	667,740,435	882,032,644	890,983,789	442,335,826	82,709,910	637,597,193

8.2 Trace Periods: Traffic Type-Specific Hurst Estimates

The following table contains H estimates for the different classes of network traffic we analyzed during each Period.

Hurst Parameter Estimates								
		Period 1	Period 2	Period 3	Period 4	Period 5	Period 6	Period 7	Period 8
All Traffic	Packets	0.95520	0.96441	0.94868	0.85791	0.85799	0.86701	0.92651	0.95363
	Bytes	0.87642	0.96955	0.87710	0.87085	0.87137	0.85573	0.94464	0.95093
HTTP	Packets	0.94945	0.87780	0.80829	0.86990	0.83161	0.86392	0.83106	0.92036
	Bytes	0.87582	0.85648	0.82850	0.87987	0.86223	0.87063	0.87913	0.91627
FTP	Packets	0.89685	0.90562	0.91052	0.85216	0.84384	0.90293	0.95554	0.74845
	Bytes	0.88455	0.90333	0.90049	0.84905	0.83710	0.85200	0.95437	0.83282
KAZAA	Packets	0.93958	0.85491	0.98318	0.78898	0.76430	0.84411	0.82718	0.86670
	Bytes	0.83426	0.84666	0.75891	0.76551	0.72178	0.82841	0.81239	0.87403
AIM	Packets	0.86179	0.81462	0.92270	0.93367	0.90097	0.87278	0.84509	0.91522
	Bytes	0.90069	0.72204	0.94426	0.95015	0.91072	0.90382	0.86291	0.93200

8.3 Capture Scripts

We ran the following perl script on porky to obtain the data

———begin perl capture script——

```
#!/usr/bin/perl
$count=0;
while ($count<200)
{ $date=`date +%m-%d-%H-%M-%S`;
chop $date;
`tcpdump -i eth1 -c 4000000 -v -tt > cap$date`;
`gzip cap$date`;
`mv cap$date.gz /mnt/saturn`;
$count++; }
```
———end perl capture script——

This script just captures (about) 4,000,000 packets and saves it to a file whose name includes that date and time of the start time of the capture. The file then gets zipped and moved to a network drive.

The output (before compression) was similar to:

1007431864.359863 216.32.120.183.http > 141.166.228.124.2348: . 1460:2920(1460) ack 1 win 28322 (ttl 116, id 72, len 1500)

1007431864.359863 141.166.228.124.2348 > 216.32.120.183.http: . [tcp sum ok] ack 2920 win 4380 (DF) (ttl 127, id 51788 , len 40)

1007431864.359863 141.166.224.73.2015 > 152.17.90.80.5190: . [tcp sum ok] ack 588 win 17520 (DF) (ttl 127, id 39603, len 40)

8.4 Refining Scripts

8.4.1 Removing extra information

In order to refine the captures I then decompressed each .gz file and ran it through the following gawk script:

———begin test.gawk——

```
#! /usr/local/bin/gawk -f
$6 ~ /len/ {print $1 " " $2 " " $4 " " $7}
$7 ~ /len/ {print $1 " " $2 " " $4 " " $8}
$8 ~ /len/ {print $1 " " $2 " " $4 " " $9}
$9 ~ /len/ {print $1 " " $2 " " $4 " " $10}
$10 ~ /len/ {print $1 " " $2 " " $4 " " $11}
```

```
$11 ~ /len/ {print $1 " " $2 " " $4 " " $12}
$12 ~ /len/ {print $1 " " $2 " " $4 " " $13}
$13 ~ /len/ {print $1 " " $2 " " $4 " " $14}
$14 ~ /len/ {print $1 " " $2 " " $4 " " $15}
$15 ~ /len/ {print $1 " " $2 " " $4 " " $16}
$16 ~ /len/ {print $1 " " $2 " " $4 " " $17}
$17 ~ /len/ {print $1 " " $2 " " $4 " " $18}
$18 ~ /len/ {print $1 " " $2 " " $4 " " $19}
$19 ~ /len/ {print $1 " " $2 " " $4 " " $20}
$20 ~ /len/ {print $1 " " $2 " " $4 " " $21}
$21 ~ /len/ {print $1 " " $2 " " $4 " " $22}
$22 ~ /len/ {print $1 " " $2 " " $4 " " $23}
$23 ~ /len/ {print $1 " " $2 " " $4 " " $24}
$24 ~ /len/ {print $1 " " $2 " " $4 " " $25}
```
————end test.gawk——

This script finds which field the pattern *len* is in, and then prints the timestamp of each packet in number of seconds since 00:00:00 1970-01-01 UTC, the source IP and port, and the destination IP and port, and size (in kilobytes), respectively.

Sample output was:

1007431864.359863 216.32.120.183.http 141.166.228.124.2348: 1500)
1007431864.359863 216.32.120.183.http 141.166.228.124.2348: 1500)
1007431864.359863 141.166.228.124.2348 216.32.120.183.http: 40)
1007431864.359863 152.17.90.80.5190 141.166.224.73.2015: 628)
1007431864.359863 141.166.224.73.2015 152.17.90.80.5190: 40)
1007431864.359863 141.166.188.7.ssh 65.97.30.64.3474: 104)
1007431865.709863 141.166.226.219 207.68.177.125: 68,

We used a gawk script to eliminate commas ',' and right parentheses ')'. This was done in the command line for each trace in each time period. One of these commands would thus look like:

```
gzip -dc cap12-03-20-48-18.gz | test.gawk | \
  gawk '{sub(/[),]/,"");print}' | gzip -cvf - > P3-1.gz
```

8.4.2 Printing desired information

I was now ready to just take each packet's timestamp and size. Thus, we need to print the 1st and the 4th field of 8.4.1's output:

gzip -dc P3-1.gz | gawk '{print $1"\t"$4}'

If however, you wanted to look only at http traffic, you would run:

gzip -dc P3-1.gz | grep http | gawk '{print $1"\t"$4}'

You could search for ftp packets in a similar fashion, but if you are looking for P2P traffic, you need to find the port number for the service and then look for that number in the second or third column. A gawk script that would acheive this (and prints the 1st and 4th field) is:

————-begin kazaa.gawk————-

```
#! /usr/local/bin/gawk -f
$2 ~ /.1214/ {print $1 "\t" $4}
$3 ~ /.1214/ {print $1 "\t" $4}
```

————-end kazaa.gawk————-

The command looks like:

gzip -dc P3-1.gz | kazaa.gawk

Similar scripts were written for AIM traffic by replacing *1214* with *5190*. The output of 8.4.2 is similar to:

1007431864.359863 1500
1007431864.359863 1500
1007431864.359863 40
1007431864.359863 628
1007431864.359863 40

8.4.3 Removing undesired information (again)

Tcpdump gave us extra information and each packet record was variable length, thus we did not see certain special cases earlier because they only occur once every 1,000 or 10,000 packets. The special cases were:

WARNING: Short Try

1007431865.709863 141.166.226.219 207.68.177.125: RA

We don't want any lines that have any character that is not a number in the last field. The following script acheives this:

———-begin noA.gawk———-
```
#! /usr/local/bin/gawk -f
{ if( $NF !~ /[^0-9]/ ) print }
```
———-end noA.gawk———-

So far we have:
gzip -dc P3-1.gz | gawk '{print $1"\t"$4}' | noA.gawk
gzip -dc P3-1.gz | grep http | gawk '{print $1"\t"$4}' | noA.gawk
gzip -dc P3-1.gz | kazaa.gawk | gawk '{print $1"\t"$4}' | noA.gawk

Our output for each of them is still similar to:
1007431864.359863 1500
1007431864.359863 1500
1007431864.359863 40
1007431864.359863 628
1007431864.359863 40

8.4.4 Counting Packets and Bytes

We are finally ready to count the number of packets and bytes in each 10 ms interval. I wrote a C program that would count the number of occurences of 1007431864.35 (for example), and added the number of bytes that is next to each occurence. The source looks like:

—-begin CONTAR.C———-
```
#include <stdio.h>
#include <ctype.h>

main()
{
  char time_old[18];
  char time_new[18];
```

```
long lc=0;
char size[8];
int total = 0;
int count = 0;
double avg  = 0;
register int i = 0;
register int c;
time_old[17]=time_new[17]='\0';
while( i<17 ){
  time_old[i] = time_new[i] = 0;
  i++;
}
i=0;

while( c=getchar() > 32 ){
  i=0;
  time_new[i] = '1';
  i++;
  while( i<10 ){
    time_new[i] = c = getchar();
    i++;
  }
  getchar();
  while( i<12 ){
     time_new[i] = c = getchar();
    i++;
  }
  while( i<16){
    getchar();
    time_new[i] = '\0';
    i++;
  }
  time_new[i]='\0';
  c = getchar();
  i=0;
  if(isdigit(c=getchar()) ){
   size[i++]=c;
```

```
    while( isdigit(c=getchar()) )
      size[i++] = c;
   }
   else{
      while(c > 32) c=getchar();
       size[i++]='4';size[i++]='0';
   }
   size[i]='\0';
   if(strncmp(time_new,time_old,13)!= 0){
    avg = (total*1.0)/count;
    if(time_old[0] != 0){
      printf("%s\t%d\t%d\n", time_old,count,total);
      total=count=0;
      lc++;
    }
    strcpy(time_old,time_new);
   }

   count++;
   total += atoi(size);
   i=0;
  }
   avg = (total*1.0)/count;
   printf("%s\t%d\t%d\n", time_old,count,total);
}
```

—-end CONTAR.C ————————

This file is very dependent on the input it expects to receive. In general this is not the best solution, it was not meant to exhibit the best software engineering principles. It is, however, a practical one for this application that we believe is time-efficient.

Sample output for the following input

1007430498.759863 1216
1007430498.759863 40
1007430498.769863 1500
1007430498.769863 40
1007430498.769863 576

1007430498.769863 384
1007430498.769863 48
1007430498.789863 40
1007430498.789863 1500
1007430498.799863 1500
1007430498.799863 1216
1007430498.799863 1500
1007430498.799863 1500
1007430498.799863 1216
1007430498.819863 40
1007430498.819863 40

is:
100743049875 2 1256
100743049876 5 2548
100743049878 2 1540
100743049879 5 6932
100743049881 2 80

Our output is of the form:
10-ms-interval packets bytes

Unfortunately, we have overlooked something.

8.4.5 Filling in the gaps

What we overlooked was that CONTAR.C does not look for increments of 10 ms, it just reads the next line of input and compares the timestamps up to the 10 ms slot of the character array to see if the time has changed. The numbers were large so I used character arrays instead of numbers, and I did not catch this error until after writing a succesful CONTAR.C. I therefore decided it would be easier (and safer) to write yet another program that would convert each timestamp character array to a long number and then fill in each 10 ms interval with no activity. The program FILL.C accomplishes this.

————begin FILL.C————
#include <stdio.h>

```
#include <ctype.h>
#include <stdlib.h>

main()
{
  int MAX = 63;
  char line[MAX];
  char oldline[MAX];
  char time[14];
  char oldtime[14];
  char size[MAX];
  char oldsize[MAX];
  int length = 0;
  char c='4';
  int i,j,k=0;
  int SIZE , OLDSIZE,  F , TOTAL=0;
  ulong TIME, OLDTIME =0;
  int dist = 0;
  i=j=0;
  getline(oldline,63);
  i++;i++;
  while(oldline[i] >= '0') oldtime[j++]=oldline[i++];
  oldtime[j]='\0';
  OLDTIME = atol(oldtime);
  printf("%d\t",OLDTIME);
  while(oldline[i++] != '\0') printf("%c",oldline[i]);
  while( getline(line,63) ){
    i=j=0;k=1;
    i++;i++;
    while(line[i] >= '0') time[j++] = line[i++];
    time[j]='\0';
    TIME = atol(time);
    dist = TIME-OLDTIME;
    for(k=1; k < dist; k++){
      OLDTIME++;
      printf("%d\t0\t0\n",OLDTIME);
    }
```

```
    printf("%d\t",TIME);
    while(line[i++] != '\0') printf("%c",line[i]);
    strcpy(oldtime,time);
    OLDTIME = atol(oldtime);
  }
}

 /* The following program is from p. 29 in [19] */
int getline(char s[], int lim)
{
  int c,i;

  i=0;
  while(-lim > 0 && (c = getchar()) != EOF && c != '\n')
    s[i++] = c;
  if (c == '\n')
    s[i++] = c;
  s[i] = '\0';
  return i;
}
```

----------end FILL.C----------

We needed to get rid of the first few characters in order to stay within the limits of LONG_MAX as defined in [19]. Those digits we removed did not change in a given Period, thus losing them does not affect our calculations. Feeding FILL the output from 8.4.4, we get:

743049875 2 1256
743049876 5 2548
743049877 0 0
743049878 2 1540
743049879 5 6932
743049880 0 0
743049881 2 80

We are now ready to start R/S analysis!
But first, let's recap. So far we have:
tcpdump -i eth1 -c 4000000 -v -tt > cap$date

```
gzip -dc cap12-03-20-48-18.gz | test.gawk | \
  gawk '{sub(/[),]/,"");print}' | gzip -cvf - > P3-1.gz
gzip -dc P3-1.gz | gawk '{print $1"\t"$4}' | \
  noA.gawk | CONTAR | FILL > P3-1.ref.1
gzip -dc P3-1.gz | grep http | gawk '{print $1"\t"$4}' | \
  noA.gawk | CONTAR | FILL > P3-1.http
gzip -dc P3-1.gz | kazaa.gawk | noA.gawk | CONTAR | \
  FILL > P3-1.kazaa
```

8.4.6 R/S Analysis

We now must strip only the packet counts or byte counts in order to proceed to R/S analysis. If we wanted the packets obtained from KaZaA traffic during Period 8, and the bytes obtained from all traffic from Period 3, we could execute:

```
cat P3-?.kazaa | gawk '{print $2}' > P3.kazaa.packets.dat
cat P3-?.ref.1 | gawk '{print $3}' > P3.bytes.dat
```

Now we are ready to use a slightly modified version the *hurst.c* program found in [16].

```
----------begin hurst.c----------
//=Program to compute the R/S statistic for a series Xof size N
//=> Used to estimate the self-similarity Hurst parameter (H)
//= Notes:
//= 1) Input from input file "in.dat" to stdin (see example below)
//=       * Comments are bounded by "&" characters at
//=         the beginning and end of the comment block
//=   2) Output is to stdout
//=   3) X should have a "large number" of values for a
//=     "good" R/S value to be computed.
//= Example "in.dat" file:
//=
//=   & Sample series of data which can be integers or reals.
//=     There are 6 values in this file. &
//=   12
//=   56
//=   99
```

```
//=  111
//=   87
//=   99
//= Example output (for above "in.dat"):
//=
//= ---------- hurst.c ---------
//=   R/S = 2.557638 for series X of 6 values
//=  ------------------------
//= Build: bcc32 hurst.c, cl hurst.c, gcc hurst.c -lm
//= Execute: hurst2 < in.dat
//= Author: Kenneth J. Christensen
//=         University of South Florida
//=         WWW: http://www.csee.usf.edu/~christen
//=         Email: christen@csee.usf.edu
//= History: KJC (09/16/98) - Genesis
//=         KJC (09/05/00) - Removed inside double loop
//=                          per advice from Sunwoo Lee.
//=                          Program runs much faster.
//=         KJC (10/31/00) - Fixed a one-off error in main
//=                          loop.  See "Fix #1" tag.

//--- Include files ------------------------------
#include <stdio.h>              // Needed for printf() and feof()
#include <math.h>               // Needed for pow()
#include <stdlib.h>             // Needed for exit() and atof()
#include <string.h>             // Needed for strcmp()

//--- Defines ------------------
// Maximum size of time series data array
#define MAX_SIZE 10000000L

//--- Globals ------------------
double   X[MAX_SIZE];       // Time series read from "in.dat"
long int N;                 // Number of values in in.dat

//--- Function prototypes ------
void  load_X_array(void);   // Load X array from "in.dat"
```

```
double compute_rs(void);      // Compute R/S for X of length N

//= Main program
void main(void)
{
  double  rs_value;              // Computed R/S value

  // Load the series X
  //printf("------------------- hurst.c ---\n");
  load_X_array();

  // Compute R/S value for series X of length N
  rs_value = compute_rs();

  // Output R/S value
  printf("%ld\t%f\n",N,rs_value);

}

//= Function to load X array from stdin and determine N
void load_X_array(void)
{
  char      temp_string[1024];     // Temporary string variable

  // Read all values into X
  N = 0;
  while(1)
  {
    scanf("%s", temp_string);
    if (feof(stdin)) goto end;

    // This handles a comment bounded by "&" symbols
    while (strcmp(temp_string, "&") == 0)
    {
      do
      {
        scanf("%s", temp_string);
```

```
      if (feof(stdin)) goto end;
    } while (strcmp(temp_string, "&") != 0);
    scanf("%s", temp_string);
    if (feof(stdin)) goto end;
   }

   // Enter value in array and increment array index
   X[N] = atof(temp_string);
   N++;

   // Check if MAX_SIZE data values exceeded
   if (N >= MAX_SIZE)
   {
    printf("*** ERROR - greater than %ld data values \n",
       MAX_SIZE);
    exit(1);
   }
  }

  // End-of-file escape
  end:

  return;
}

//=  Function to compute R/S value for series X of length N
double compute_rs()
{
  double    mom1;     // First moment
  double    mom2;     // Second moment
  double    x_bar;    // Mean (X_bar value)
  double    s;        // Standard deviation (S value)
  double    w;        // W value
  double    r;        // R value
  double    min_w;    // Minimum W value
  double    max_w;    // Maximum W value
  double    rs_value; // R/S value to be returned
```

```
  double    sum;       // Temporary sum value
  long int  i, j;      // Loop counters

  // Loop to compute mean and standard deviation of X
  mom1 = mom2 = 0.0;
  for (i=0; i<N; i++)
  {
    mom1 = mom1 + (X[i] / N);
    mom2 = mom2 + (pow(X[i], 2.0) / N);
  }
  x_bar = mom1;
  s = sqrt(mom2 - pow(mom1, 2.0));

  // Double loop to find minimum and maximum W values
  min_w = max_w = 0.0;
  sum = 0.0;
  for (i=0; i<N; i++)
  {
    sum = sum + X[i];

    w = sum - ((i+1) * x_bar);        // Fix #1
    if (w > max_w) max_w = w;
    if (w < min_w) min_w = w;
  }

  // Compute R value as maximum W minus minimum W
  r = max_w - min_w;

  // Compute R/S value
  rs_value = r / s;

  // Return R/S value
  return(rs_value);
}
```

————————end hurst.c————————

Thus, we ran the following commands:

hurst < P3.kazaa.packets.dat > P3.kazaa.packets.hurst.txt

The file *P3.kazaa.packets.hurst.txt* contains the value of N (the size of the set of numbers) and the corresponding R/S estimate:

558404 126310.199372

8.4.7 Estimating H

In order to obtain an estimate H we need to aggregate the packet/byte counts and obtain a few R/S estimates for different values of N. To do this, we use a modified version of *block.c*, also from [16].

```
———-begin block.c———
//= Program to block a time series X into block means
//= Notes:
//=   1) Input from input file "in.dat" to stdin
//=     * Comments are bounded by "&" characters at the
//=       beginning and end of the comment block
//=   2) The block size M is in the #define section
//=   3) If mod(N,M) is not zero, then the last remainder values
//=     are notblocked
//=   4) Output is to stdout
//=————————————————————-
//= Example "in.dat" file:
//=
//=  & Here is a series of 6 values to be blocked with M = 2  &
//=  21
//=  3
//=  55
//=  45
//=  12
//=  5
//=————————————————————-
//= Example output (for above "in.dat" and M = 2):
//=
//=  & ———————————— block.c —— &
//=    12.000000
```

```
//=    50.000000
//=    8.500000
//=  &  Output 3 block means for a block size of 2
//=  ------------------------------ &
//=-----------------------------------
//=  Build: gcc block.c, bcc32 block.c, cl block.c
//=-----------------------------------
//=  Execute: block < in.dat
//=-----------------------------------
//=  Author: Kenneth J. Christensen
//=          University of South Florida
//=          WWW: http://www.csee.usf.edu/~christen
//=          Email: christen@csee.usf.edu
//=-----------------------------------
//=  History: KJC (10/02/98) - Genesis
//=         KJC (02/24/99) - Fixed a compile error
//=         KJC (06/31/99) - Fixed error with not finishing series

//---Include files-------------------------
#include <stdio.h>              //Needed for printf() and feof()
#include <stdlib.h>             //Needed for exit() and atof()
#include <string.h>             //Needed for strcmp()

//---Defines-------------------------------
//Max size of time series data array
#define MAX_SIZE 10000000L
//Blocking size
#define M           10L

//---Globals-------------------------------
double    X[MAX_SIZE];      // Time series read from "in.dat"
long int  N;                // Number of values in X[]

//---Prototypes----------------------------
void  load_X_array(void);       // Load X array
```

```
//=  Main program

void main(void)
{
 long int  count;               // Count of number of blocks
 double    sum;                 // Temporary sum variable
 double    block_mean;          // Compute block mean
 long int  i, j;                // Loop counters

 // Load the series X
 //printf("& ---------- block.c ------ & \n");
 load_X_array();

 // Compute and output block means
 //    (aggregated blocks of size M)
 count = 0;
 for (i=0; i<N; i=i+M)
 {
  sum = 0.0;
  for (j=i; j<(i + M); j++)
  {
   if (j >= N) goto end;
   sum = sum + X[j];
  }

  count = count + 1;
  block_mean = sum / M;
  printf("%f \n", block_mean);
 }

 // End of series escape
 end:

 // Output closing message
 printf("& Output %ld block means for a block size of %ld \n",
     count, M);
```

```
}

//=  Function to load X array from stdin and determine N   =
void load_X_array(void)
{
 char     temp_string[1024];     // Temporary string variable

 // Read all values into X
 N = 0;
 while(1)
 {
  scanf("%s", temp_string);
  if (feof(stdin)) goto end;

  // This handles a comment bounded by "&" symbols
  while (strcmp(temp_string, "&") == 0)
  {
   do
   {
    scanf("%s", temp_string);
    if (feof(stdin)) goto end;
   } while (strcmp(temp_string, "&") != 0);
   scanf("%s", temp_string);
   if (feof(stdin)) goto end;
  }

  // Enter value in array and increment array index
  X[N] = atof(temp_string);
  N++;

  // Check if MAX_SIZE data values exceeded
  if (N >= MAX_SIZE)
  {
   printf("*** ERROR - greater than %ld data values \n",
      MAX_SIZE);
   exit(1);
  }
```

```
  }

  // End-of-file escape
  end:

  return;
}
```
———-end block.c———-

We compiled this file and saved the output code to a file named *block10*. Then we changed the blocking size M to 5, recompiled, and saved the output to a file named *block5*. We can now run an analyze.sh script:

———-begin analyze.sh———-
```
block5 < $1".dat" > $1".dat5"
block10 < $1".dat" > $1".dat10"
block5 < $1".dat5" > $1".dat25"
block10 < $1".dat10" > $1".dat100"
block5 < $1".dat25" > $1".dat125"
block5 < $1".dat125" > $1".dat625"
block10 < $1".dat100" > $1".dat1000"
block 5 < $1".dat625" > $1".dat3125"
block10 < $1".dat1000" > $1".dat10000"
hurst < $1".dat" > $1".hurst.txt"
hurst < $1".dat5" >> $1".hurst.txt"
hurst < $1".dat10" >> $1".hurst.txt"
hurst < $1".dat25" >> $1".hurst.txt"
hurst < $1".dat100" >> $1".hurst.txt"
hurst < $1".dat125" >> $1".hurst.txt"
hurst < $1".dat625" >> $1".hurst.txt"
hurst < $1".dat1000" >> $1".hurst.txt"
hurst < $1".dat3125" >> $1".hurst.txt"
hurst < $1".dat10000" >> $1".hurst.txt"
```
———-end analyze.sh———-

Thus, we would execute:
analyze.sh P3.kazaa.packets

analyze.sh P3.bytes

The contents of the *P3.kazaa.packets.hurst.txt* file that is produced are:

558404 126310.199372
111680 28487.475111
55840 14636.914452
22336 5997.506536
5584 1530.838622
4467 1226.322975
893 246.440876
558 153.981745
89 25.168483
55 14.997397

Which is just a set of $N's$ with their corresponding R/S estimates. Now we need to fit a line through these points.

8.4.8 Least-Squares Line Fit

We used following header file and made some changes to *linreg.cpp* so that it first sets $x = Log(x)$ and $y = Log(y)$ before fitting the line. Both of the original files can be found in [17].

———begin linreg.h———

```
/* linreg.h */
#ifndef _LINREG_H_
#define _LINREG_H_
#include <iostream.h>

// a class encapsulating a point in Cartesian coordinates
class Point2D
{
   public:
      Point2D(double X = 0.0, double Y = 0.0) : x(X), y(Y)
      { }

      void setPoint(double X, double Y) { x = X; y = Y; }
```

```
        void setX(double X) { x = X; }
        void setY(double Y) { y = Y; }

        double getX() const { return x; }
        double getY() const { return y; }

    private:
        double x, y;
};

// a linear regression analysis class
class LinearRegression
{
    friend ostream& operator< <(ostream&, LinearRegression&);

    public:
        // Constructor using an array of Point2D objects
        // This is also the default constructor
        LinearRegression(Point2D *p = 0, long size = 0);

        LinearRegression(double *x, double *y, long size = 0);

virtual void addXY(const double& x, const double& y);
        void addPoint(const Point2D& p)
        { addXY(p.getX(), p.getY()); }

        // Must have at least 3 points to calculate
        // standard error of estimate.
        //Do we have enough data?
        int haveData() const { return (n > 2 ? 1 : 0); }
        long items() const { return n; }

virtual double getA() const { return a; }
virtual double getB() const { return b; }

        double getCoefDeterm() const  { return coefD; }
        double getCoefCorrel() const { return coefC; }
```

```
        double getStdErrorEst() const { return stdError; }
virtual double estimateY(double x) const
        { return (a + b * x); }

    protected:
        long n;             // number of data points input
        double sumX, sumY;  // sums of x and y
        double sumXsquared, // sum of x squares
               sumYsquared; // sum y squares
        double sumXY;       // sum of x*y

        double a, b;        // coefficients of f(x) = a + b*x
        double coefD,       // coefficient of determination
               coefC,       // coefficient of correlation
               stdError;    // standard error of estimate

        void Calculate();   // calculate coefficients
};

#endif                      // end of linreg.h
```

———-end linreg.h———-

———-begin linreg.cpp——-

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
#include "linreg.h"
void main()
{
    int pts = 10;
    int j,i;
    double x[pts], y[pts];
    i=j=0;
    while(j<pts){
     cin > > x[j] > > y[j];
     j++;
    }
```

```
    for(i=0;i<j;i++){
      x[i] = log(x[i]);
      y[i] = log(y[i]);
    }
    LinearRegression lr(x, y, pts);
    cout << "Number of x,y pairs = " << lr.items() << endl;
    cout << lr << endl;
    cout << "Coefficient of Determination = "
        << lr.getCoefDeterm() << endl;
    cout << "Coefficient of Correlation = "
        << lr.getCoefCorrel() << endl;
    cout << "Standard Error of Estimate = "
        << lr.getStdErrorEst() << endl;
}
```

———-end linreg.cpp———

We compiled *linreg.cpp* and saved the output to *fitline_new* and then ran:
fitline_new < P3.kazaa.packets.hurst.txt

And the output produced is:
Number of x,y pairs = 10
*f(x) = -1.18099 + (0.983177 * x)*
Coefficient of Determination = 0.999835
Coefficient of Correlation = 0.999918
Standard Error of Estimate = 0.0409876

Our estimate for H is therefore 0.983177, and our is R^2 value is 0.999918.

8.4.9 Sum of Packets and Bytes of a Period

Of course, finding out the total number of bytes and packets in a particular Period is useful. For this, we used summary1.c from [16].

———begin summary1.c———

```
//= Program to compute summary statistics for a series X of size
N
//= - Computes min, max, sum, mean, var, std dev, and cov
```

```
//= Notes:
//=    1) Input from input file "in.dat" to stdin (see example
below)
//=       * Comments are bounded by "&" characters at the
//=         beginning and end of the comment block
//=    2) Output is to stdout
//= Example "in.dat" file:
//=  & Sample series of data which can be integers or reals.
//=    There are 11 values in this file. &
//=   50
//=   42
//=   48
//=   61
//=   60
//=   53
//=   39
//=   54
//=   42
//=   59
//=   53
//= Example output (for above "in.dat"):
//=
//=————————— summary1.c ————————-
//=    Total of 11 values
//=      Minimum  = 39.000000 (position = 6)
//=      Maximum  = 61.000000 (position = 3)
//=      Sum      = 561.000000
//=      Mean     = 51.000000
//=      Variance = 52.545455
//=      Std Dev  = 7.248824
//=      CoV      = 0.142134
//=————————————————————————
//=  Build: gcc summary1.c -lm, bcc32 summary1.c, cl sum-
mary1.c
//=————————————————————————
//= Execute: summary1 < in.dat
//=————————————————————————
```

```
//= Author: Kenneth J. Christensen
//=         University of South Florida
//=         WWW: http://www.csee.usf.edu/~christen
//=         Email: christen@csee.usf.edu
//=-------------------------
//= History: KJC (05/23/00) - Genesis

//--- Include files ------------
#include <stdlib.h>        // Needed for exit() and atof()
#include <string.h>        // Needed for strcmp()
#include <math.h>          // Needed for pow()

//--- Defines ---------------
#define MAX_SIZE 10000000L  // Maximum size of time series
data array

//--- Globals ---------------
double    X[MAX_SIZE];    // Time series read from "in.dat"
long int  N;              // Number of values in "in.dat"

//--- Function prototypes ---
void  load_X_array(void);    // Load X array

//= Main program
void main(void)
{
 double   min, max;          // Minimum and maximum values
 long int minpos, maxpos;    // Positions of min and max
 double   sum;               // Sum of values
 double   mom1, mom2;        // First and second moments of
values
 double   mean;              // Computed mean value
 double   var;               // Computed variance
 double   stddev;            // Computed standard deviation
 double   cov;               // Computed coefficient of variation
 long int i;                 // Loop counter
```

```
// Load the series X
printf("————————— summary1.c ——\n");
load_X_array();

// Loop to compute min, max, sum,
//     first moment (mean), and second moment
min = max = X[0];
minpos = maxpos = 0;
sum = mom1 = mom2 = 0.0;
for (i=0; i<N; i++)
{
  if (X[i] <= min)
  {
    min = X[i];
    minpos = i;
  }
  if (X[i] >= max)
  {
    max = X[i];
    maxpos = i;
  }
  sum = sum + X[i];
  mom1 = mom1 + (X[i] / N);
  mom2 = mom2 + (pow(X[i], 2.0) / N);
}

// Compute mean, variance, standard deviation, and cov
mean = mom1;
var = mom2 - pow(mom1, 2.0);
stddev = sqrt(var);
cov = sqrt(var) / mom1;

printf("  Total of %ld values \n", N);
printf("    Minimum  = %f (position = %ld) \n", min, minpos);
printf("    Maximum  = %f (position = %ld) \n", max, maxpos);
printf("    Sum      = %f \n", sum);
printf("    Mean     = %f \n", mean);
```

```
  printf("    Variance = %f \n", var);
  printf("    Std Dev  = %f \n", stddev);
  printf("    CoV      = %f \n", cov);
  printf("————————————————\n");
}

//= Function to load X array from stdin and determine N
void load_X_array(void)
{
  char     temp_string[1024];    // Temporary string variable

  // Read all values into X
  N = 0;
  while(1)
  {
    scanf("%s", temp_string);
    if (feof(stdin)) goto end;

    // This handles a comment bounded by "&" symbols
    while (strcmp(temp_string, "&") == 0)
    {
      do
      {
        scanf("%s", temp_string);
        if (feof(stdin)) goto end;
      } while (strcmp(temp_string, "&") != 0);
      scanf("%s", temp_string);
      if (feof(stdin)) goto end;
    }

    // Enter value in array and increment array index
    X[N] = atof(temp_string);
    N++;

    // Check if MAX_SIZE data values exceeded
    if (N >= MAX_SIZE)
    {
```

```
      printf("*** ERROR - greater than %ld data values \n",
          MAX_SIZE);
      exit(1);
    }
  }

  // End-of-file escape
  end:

  return;
}
```
———end summary1.c———

References

[1] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the Self-Similar Nature of Ethernet Traffic", *IEEE/ACM Trans. Networking*, vol. 2, no. 1, pp. 1-15, 1994.

[2] V. Paxon and S. Floyd, "Wide-area traffic: The Failure of Poisson Modeling," *IEEE/ACM Trans. Networking*, vol. 3, no. 3, pp. 226-244, 1995.

[3] "Dictionary.com/burst", Dictionary.com, 2002. Lexico LLC. http://www.dictionary.com

[4] M. E. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes", *IEEE/ACM Trans. Networking*, vol. 5, no. 6, pp. 835-846, 1997.

[5] "Application Performance Solutions from Packeteer", 2002. http://www.packeteer.com

[6] "KaZaA Media Desktop", 2002. http://www.kazaa.com/en/index.htm

[7] Seung Hoon Hong, Rae-Hong Park, and Chang Bum Lee, "Hurst Parameter Estimation of Long-Range Dependent VBR MPEG Video Traffic Jin ATM Networks", *Journal of Visual Communication and Image Representation*, vol. 12, no. 1, pp. 44-65, 2001.

[8] "AOL Instant Messenger (TM)", 2002. http://www.aim.com/index.adp

[9] J. Cao, W. S. Cleveland, D. Lin, and D. X. Sun. "The Effect of Statistical Multiplexing on Internet Packet Traffic: Theory and Empirical Study", Technical Report, Bell Labs, 2001.

[10] "Snort - The Open Source Network IDS", 2002. http://www.snort.org

[11] "The Ethereal Network Analyzer", 2002. http://www.ethereal.com

[12] "TCPDUMP public repository", 2002. http://www.tcpdump.org

[13] B. Tsybakov and N. Georganas, "On SelfSimilar Traffic in ATM Queues: Definitions, Overflow Probability and Cell Delay Distribution", IEEE/ACM Transactions on Networking, vol. 5, no. 3, pp. 397-409, 1997.

[14] T. Hagiwara, H. Doi, H. Tode, and H. Ikeda, "The High Speed Calculation Method of the Hurst Parameter Using Real Network Traffic", *Proceedings from the 25th Annual IEEE Conference on Local Computer Networks (LCN 2000)*, pp. 662-669.

[15] K.J. Christensen, "Reproduction of some key results in Leland et al.", 2001. University of South Florida. http://www.csee.usf.edu/~christen/tools/bellcore.pdf

[16] K.J. Christensen, "Tools page for Kenneth J. Christensen", 1998. http://www.csee.usf.edu/~christen/toolpage.html

[17] David C. Swaim II, "A Simple Linear Regression Class", 2000. http://www.cuj.com/articles/2000/0008/0008e/0008e.htm?topic=articles

[18] J. L. Devore, "Probability and Statistics for Engineering and the Sciences" (fifth edition), Brooks/Cole, 2000.

[19] B.W. Kernighan and D. M. Ritchie, "The C Programming Language" (second edition), Prentice Hall, Inc., 1988.