SIMULATING SELF-SIMILAR VBR TRAFFIC

MALINI SUBRAMANIAM
(WEK010151)

A Graduation Exercise submitted to the Faculty of Computer Science and Information Technology University Malaya in partial fulfillment of the requirements for the Degree of Bachelor in Computer Science

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY, UNIVERSITY MALAYA
MARCH 2004

Dedicated to my Amma and Appa

ABSTRACT

Studies on the nature of VBR traffic have shown that VBR traffic has self-similar characteristics. However, most traditional network traffic models that are used in the simulation of VBR traffic are unable to capture these self-similar qualities. As VBR traffic is forecasted to be a substantial portion of network traffic, it is imperative that the self-similarity of VBR traffic be taken into account whilst designing both network simulators and real networks.

The aim of this project is to implement a module that will simulate self-similar VBR traffic. The module will be implemented in an existing network simulator : the UMJaNetSim network simulator. Users of the simulator will then be able to generate self-similar VBR traffic.

ACKNOWLEDGEMENTS

The completion of this project would have been impossible without the guidance of my supervisor, Mr. Phang Keat Kong. I would like to thank him for patiently guiding me throughout these past few months. Special thanks to my moderator, Mr. Ang Tan Fong.

I would like to thank Mr. Nithyanandan Natchimuthu for his advice and support. Special thanks to Mr. Chow Chee Onn for his advice and guidance.

I would also like to thank my fellow group members; Wai Hong, Lee Wen, Geck Hiang, Andrew, Chin We, Yee Boon, and Kai Yan, for all their help and support.

Last but not least, I would like to express my deepest gratitude to my family for their love and encouragement and to my friends Giri and Manmeet for their support.

TABLE OF CONTENTS

ABSTRACT	*ii*
ACKNOWLEDGEMENTS	*iii*
TABLE OF CONTENTS	*iv*
LIST OF FIGURES	*x*
LIST OF TABLES	*xi*
LIST OF GRAPHS	*xi*
CHAPTER 1 : INTRODUCTION	
1.1 : Introduction	**1**
1.1.1 : Introduction to Network Simulation	**1**
1.1.2 : Introduction to VBR Video	**1**
1.2 : Motivation	**1**
1.3 : Project Objectives	**2**
1.4 : Project Scope	**2**
1.5 : Project Schedule	**3**
1.6 : Report Organization	**3**

CHAPTER 2 : LITERATURE REVIEW

2.1 : Introduction 6

2.2 : Network Simulation 6

2.2.1 : Simulation 7

2.2.1.1 : Discrete-Event Simulation 8

2.2.2 : Examples of Network Simulators 9

2.2.2.1 : NIST ATM/HFC Network Simulator 9

2.2.2.2 : SimATM Network Simulator 10

2.2.2.3 : UMJaNetSim Network Simulator 12

2.3 Network Traffic Models 12

2.3.1 : Types of Network Traffic 13

2.3.2 : Types of Network Traffic Models 13

2.3.2.1 : Renewal Models 14

2.3.2.2 : Markov Models 15

2.3.2.3 : Self-Similar Traffic Models 16

CHAPTER 3 : SELF-SIMILAR TRAFFIC

3.1 : Introduction 17

3.2 : Self-similarity 17

3.3.: Self-similar Data Traffic 19

3.3.1 : Continuous-time Definition 20

3.3.2 : Discrete-time Definition 21

3.3.3 : Hurst Parameter, H 21

3.3.4 : Long-range Dependence 22

3.3.5 : Heavy-tailed Distributions 23

3.3.6 : Spectral Density 23

3.4 : Self-similar Estimation Techniques 24

3.4.1 : R/S Plot 24

3.4.2 : Variance-time Plot 25

3.4.3 : Whittle's Estimator 26

3.5 : Examples of Self-similar Data Traffic 27

3.5.1 : Ethernet Traffic 28

3.5.2 : VBR (Variable Bit Rate) Video Traffic 31

3.5.2.1 : Paper 1 : "Analysis, Modeling and Generation of Self-Simlar VBR Video Traffic" (Garett & Willinger, 1994) 31

3.5.2.2 : Paper 2 : "Long-Range Dependence in Variable-Bit-Rate Video Traffic" (Beran et al, 1995) 32

3.6 Methods of Simulating Self-similar Traffic 33

3.6.1 : Random Midpoint Displacement (RMD) 33

3.6.2 : Fast Fourier Transformation (FFT) 33

3.6.3 : On/Off Processes 34

CHAPTER 4 : SYSTEM ANALYSIS

4.1 : Introduction 35

4.2 : Overview of UMJaNetSim Simulator 35

4.3 : Software Requirements 37

4.3.1 : Java Programming Language 37

4.3.2 : JCreator 2.5LE 39

4.4 : Minimal Hardware Requirements 40

CHAPTER 5 : SYSTEM DESIGN

5.1 : Introduction 41

5.2 : Components of UMJaNetSim 41

5.2.1 : JavaSim 42

5.2.2 : SimComponent 42

5.2.3 : SimParameter 43

5.2.4 : SimClock 43

5.2.5 : SimEvent 44

5.2.6 : Object Serialization and Load/Save Operations 44

5.3 : Simplified View of Implementation 45

5.4 : Design Issues 46

CHAPTER 6 : SYSTEM IMPLEMENTATION

6.1 : Introduction 47

6.2 : Implementation Steps 47

6.3 : Method of Generating Self-similar Traffic 48

6.3.1 : Self-similar Traffic Module in the NIST ATM/HFC Network Simulator 48

6.4 : Implementation of VBR Self-similar Traffic Module in UMJaNetSim 49

6.4.1 : Topology Used in UMJaNetSim with the VBR Self-Similar Module as the Traffic Source 52

6.4.2 : Properties of VBR Self-Similar Module 53

6.4.3 : Self-Similar Calculation Methods 55

6.5 : Conclusion 58

CHAPTER 7 : SYSTEM TESTING

7.1 : Introduction 59

7.2 : Testing with the R/S Statistic 59

7.3 : Testing Results 61

7.4 : Conclusions 65

CHAPTER 8 : CONCLUSION

8.1 : Conclusion **67**

8.2 : System's Strengths **67**

8.3 : System's Weaknesses **68**

8.4 : Future Enhancements **68**

REFERENCES ***69***

APPENDIX ***72***

LIST OF FIGURES

Figure 2.1 :Ways to study a system 7

Figure 2.2 : Simulation framework of SimATM 11

Figure 3.1 : Cantor set with five levels of recursion 19

Figure 3.2 : Pictorial "proof" of self-similarity in Ethernet traffic 30

Figure 3.3 : Self-similarity in VBR video 32

Figure 4.1 : Overall architecture of UMJaNetSim 36

Figure 4.2 : Screenshot of JCreator 2.5LE 39

Figure 5.1 : Components in UMJaNetSim 41

Figure 5.2 : Simplified View of Implementation 45

Figure 6.1 : VBR Self-Similar component in UMJaNetSim 0.49 50

Figure 6.2 : Topology in UMJaNetSim with VBR Self-Similar module as traffic source 53

Figure 6.3 : VBR Self-Similar properties (unique properties ticked) 53

Figure 6.4 : Error message that appears if Hurst parameter is not within 0.5 and 1.0 54

Figure 6.5 : Graph displaying the changing bit rate 55

Figure 6.6 : Methods involved in generating self-similar data 56

Figure A1 : 'go' Batch File Appendix

Figure A2 : cmd.exe Appendix

Figure A3 : UMJaNetSim Network Simulator Appendix

Figure A4 : Simple topology Appendix

Figure A5 : Pop-up Appendix

Figure A6 : Selecting the VBR Self-Similar Application Appendix

Figure A7 :Name request Appendix

Figure A8 : Component created Appendix

Figure A9 : Error message Appendix

Figure A10 : "Connect Mode" button Appendix

Figure A11 : "End Connect" button Appendix

Figure A12 : Selecting "Properties" Appendix

Figure A13 : Properties of vbr1 Appendix

Figure A14 : "Manage" button Appendix

Figure A15 : Route Table Appendix

Figure A16 : Route Table updated Appendix

Figure A17 : Properties of bte2 Appendix

Figure A18 : "Start" button Appendix

Figure A19 : Graph displaying changing bit rate Appendix

Figure A20 : "Pause" button Appendix

Figure A21 : "Reset" button Appendix

LIST OF TABLES

Table 7.1 : Test Results 61

Table A1 : Gantt chart for project schedule Appendix

LIST OF GRAPHS

Graph 7.1 : Test results on C program 62

Graph 7.2 : Test results on Java program, mean bit rate = 100 Mbits/s 63

Graph 7.3 : Test results on Java program, mean bit rate = 500 Mbits/s 63

Figure A8 : Component created Appendix

Figure A9 : Error message Appendix

Figure A10 : "Connect Mode" button Appendix

Figure A11 : "End Connect" button Appendix

Figure A12 : Selecting "Properties" Appendix

Figure A13 : Properties of vbr1 Appendix

Figure A14 : "Manage" button Appendix

Figure A15 : Route Table Appendix

Figure A16 : Route Table updated Appendix

Figure A17 : Properties of bte2 Appendix

Figure A18 : "Start" button Appendix

Figure A19 : Graph displaying changing bit rate Appendix

Figure A20 : "Pause" button Appendix

Figure A21 : "Reset" button Appendix

LIST OF TABLES

Table 7.1 : Test Results 61

Table A1 : Gantt chart for project schedule Appendix

LIST OF GRAPHS

Graph 7.1 : Test results on C program 62

Graph 7.2 : Test results on Java program, mean bit rate = 100 Mbits/s 63

Graph 7.3 : Test results on Java program, mean bit rate = 500 Mbits/s 63

Graph 7.4 : Test results on Java program, mean bit rate = 1000 Mbits/s 64

Graph 7.5 : Test results on Java program, mean bit rate = 2500 Mbits/s 64

CHAPTER 1 : INTRODUCTION

1.1 : Introduction

1.1.1 : Introduction to Network Simulation

Network simulation is a technique used to simulate networks with the ultimate objective of understanding and building better networks. There are many types of network simulators that have/are being developed; one such simulator is UMJaNetSim which was developed at the Faculty of Computer Science and Information Technology, University of Malaya.

1.1.2 : Introduction to VBR Video

Video services have been forecasted to be a substantial portion of emerging broadband networks. VBR video can be divided into two classes, video conferences and entertainment video. Video is a succession of regularly spaced still pictures called frames. Each frame is represented in digital format by a coding algorithm and subsequently compressed to save bandwidth.

1.2 Motivation

As mentioned in 1.1.2, VBR video will be a major part of network traffic. Statistical source models of video traffic are needed to design networks that deliver acceptable picture quality at minimum cost. Research has shown self-similarity is an inherent characteristic of VBR traffic. Traditional traffic models are unable to capture the self-similarity characteristics of network traffic. Using unsuitable traffic models will result in inaccurate simulation performance.

Simulating self-similar VBR traffic will enable a greater understanding towards the mechanics of both self-similarity and VBR traffic.

1.3 Project Objectives

The objective of this project is to implement a VBR traffic module into an existing network simulator, UMJaNetSim that uses a self-similar traffic generation method. In order to do this, various research has to be conducted; on the concepts of self-similarity, on the characteristics of VBR traffic, on the structure of the network simulator, UMJaNetSim and on how to implement a module into the simulator.

The ultimate aim of this entire project is to study and understand the concept of self-similarity in today's network traffic. Implementation of a module will also help one understand the workings of a network simulator.

1.4 Project Scope

The scope of the project is as follows :

(i) Develop a VBR traffic module that generates self-similar traffic in an existing network simulator

(ii) Allow the user to input the input data rate and self-similarity parameter, *H*

(iii) Show the simulation results – the output data rate

(iv) Test the outputs to see if they are indeed self-similar by way of self-similar estimation techniques

1.5 Project Schedule

The development of the entire project consists of the following tasks :

(i) Literature Review

(ii) System Analysis

(iii) System Design

(iv) System Coding/Implementation

(v) System Testing/Evaluation

(vi) Documentation

The project schedule is depicted in a Gantt chart at the Appendix (Table A1).

1.6 Report Organization

Chapter 1 is the introductory chapter. The first part presents a very brief introduction on network simulation and VBR traffic. The rest of the chapter consists of the project motivation, project objectives, project scope, project schedule and report organization.

Chapter 2 is the literature review. The concept of simulation and discrete-event simulation is viewed. A few examples of network simulators that have been developed are listed. Section 2.3 lists the types of traffic in today's networks. This section also lists the types of network traffic models used to simulate network traffic.

The third chapter deals with the main subject of this project, self-similar traffic. The first section in this chapter is a short introduction, and this is followed by a definition of self-similarity. Section 3.3 contains important terms that are frequently used when speaking in terms of self-similar data traffic. The following section lists three self-similar estimation techniques. Section 3.5 discusses three studies conducted on Ethernet and VBR video traffic that have shown that these traffic types are self-similar in nature. The last section lists a few methods for simulating self-similar traffic.

Chapter 4 covers the System Analysis. The first section is an introduction to the chapter. Section 4.1 is an overview of UMJaNetSim, the simulator into which the VBR traffic module will be implemented. Sections 4.3 and 4.4 list the software and hardware requirements, respectively.

The next chapter, Chapter 5, is the System Design. Section 5.1 is an introduction to the chapter. Section 5.2 discusses the components in UMJaNetSim. A simplified view of the implementation of the VBR module is given in Section 5.3, while the last section, Section 5.4, looks at two important design issues.

Chapter 6 is discusses the implementation of the VBR self-similar module into the system. It lists the steps taken towards the implementation (6.2), the chosen method for generating self-similar traffic (6.3), the implementation into UMJaNetSim itself and ends with a conclusion on the implementation phase (6.5).

Chapter 7 is on system testing. The first part of the chapter is a brief introduction and this is followed by a discussion on the testing method used. Section 7.3 lists the test results and Section 7.4 concludes the testing.

The last chapter, Chapter 8, lists the overall conclusion of the project as well as the system's strengths and weaknesses. A list of future enhancements is also given.

The Appendix contains the Gantt chart for the project schedule and a user manual. The user manual lists the steps that need to be taken to run the simulator and self-similar module.

CHAPTER 2 : LITERATURE REVIEW

2.1 : Introduction

This chapter is divided into three parts ; a brief introduction, a section on network simulation and a section on network traffic models.

Section 2.2 of this chapter is about network simulation. It looks at discrete-event simulation and cites a few examples of network simulators.

Section 2.3 lists the types of traffic in today's networks as well the different types of models that can be used to simulate traffic.

2.2 : Network Simulation

Telecommunications and computer networks have become the basis of the world's economic and scientific infrastructure (Hlavacs *et al*, 1999). Although the speed of networks is growing faster and faster, there are still problems in sending data from one terminal or computer to another as bottlenecks and congestion occur. It is therefore imperative that careful planning is done when installing or upgrading large networks. Simulation is a very important technique that can be used in planning the capacity of networks.

2.2.1 : Simulation

Simulation is one of the most widely used operations-research and management science techniques (Kelton & Law, 2000). It is a technique that involves computers simulating operations of real-world facilities. The real-world facilities are usually called systems.

Figure 2.1 (Kelton & Law, 2000) maps out the different ways to study a system. It can be seen that simulation is mathematical model used to experiment with a model of the system.

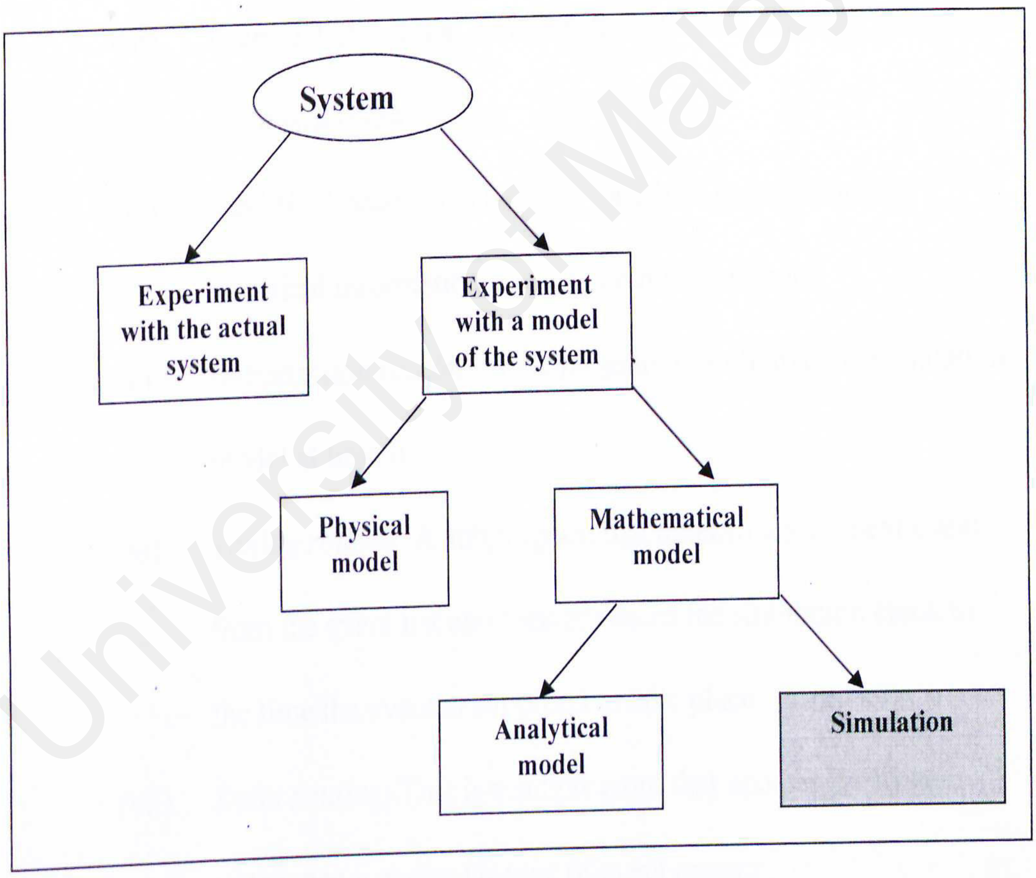

System
Experiment with the actual system
Experiment with a model of the system
Physical model
Mathematical model
Analytical model
Simulation

Figure 2.1 : Ways to study a system

Systems can be categorized as either discrete or continuous. In a discrete system, the state variables change instantaneously at separate points in time, whereas in a continuous system, the state variables change continuously with respect to time.

2.2.1.1 : Discrete-Event Simulation

Discrete-Event Simulation is concerned with the modeling of the system as it evolves over time. Discrete-event simulation models have been applied to various real-world situations, and all these models have a number of common components :

(i) *System state*: This is the collection of state variables that are necessary to describe the system at a particular time

(ii) *Simulation clock*: This is a variable that gives the current value of the simulated time

(iii) *Event list*: A list that contains the next time when each type of event will occur

(iv) *Statistical counters*: These are variables used for storing statistical information about system performance

(v) *Initialization routine*: A subprogram to initialize the simulation model at time 0

(vi) *Timing routine*: A subprogram that determines the next event from the event list and then advances the simulation clock to the time the event is supposed to take place

(vii) *Event routine*: This is a subprogram that updates the system state when a particular type of event occurs

(viii) *Library routines*: This is a set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation model

(ix) *Report generator*: A subprogram that computes estimates of the desired measures of performance and produces a report when the simulation ends. The estimates are taken from statistical counters

(x) *Main program*: This is a subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program checks for termination and invokes the report generator when the simulation is over

2.2.2 : Examples of Network Simulators

2.2.2.1 : NIST ATM/HFC Network Simulator

This simulator was developed at the National Institute of Standards and Technology (NIST) to provide a flexible testbed for both studying and evaluating the performance of ATM (Asynchronous Transfer Mode) and HFC (Hybrid Fiber Coax) networks. This tool is based on a network simulator developed at the Massachusetts Institute of Technology (in the United States of America) that provides support for discrete-event simulation techniques and has graphic user interface (GUI) capabilities.

NIST has developed this tool using both the C programming language and the X Window System running on a UNIX platform. It gives the user an interactive modeling environment with a graphical user interface.

The ATM/HFC Network Simulator allows the user to:

(i) Create different network topologies

(ii) Set the parameters of component operation

(iii) Save/load the different simulated configurations

While the simulation is running, various instantaneous performance measures can be displayed in graphical or text form on the screen or saved to files for subsequent analysis.

2.2.2.2 : SimATM Network Simulator

SimATM is an ATM (Asynchronous Transfer Mode) network simulation environment. Its aim is to provide researchers and designers of networks with a tool for teaching, research, analysis and design of ATM networks (Alberti *et al*, 1998). SimATM utilizes an event-driven simulation technique to achieve ATM simulation at the cell level. It was developed in C++ to the Windows 95/NT^{TM} operating system.

This simulator has extensive simulation statistical data gathering, which allows the simulations analysis even in network transitory or stationary state. It enables the direct comparison of simulation results with queuing system models of the queuing theory.

Figure 2.2 (Alberti et al, 1998) shows the simulation framework SimATM. The simulator was developed using an event-driven simulation technique. SimATM's kernel has a command interpreter, an event queue, an event manager and several instances of ATM networks. The command interpreter executes commands over many ATM networks, but only one ATM network can be simulated at any one time.

The event queue contains the events that are waiting to be executed. These events are taken by the event manager from the event queue and sent to their destination equipment or application blocks inside the network under simulation. These application blocks are able to follow events received from the event manager to the destination element's layer or its associated queuing system, thus ending the event's life cycle in the simulator. Simulation continues until it reaches the predetermined maximum simulation time, or until there are no more events left to be executed in the network's event queue.

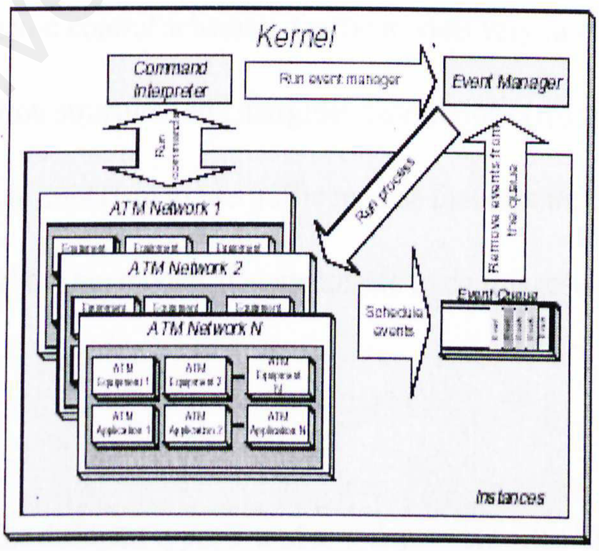


Figure 2.2 : Simulation framework of SimATM

2.2.2.3 : UMJaNetSim Network Simulator

Developed at the Faculty of Computer Science and Information Technology of University Malaya, UMJaNetSim is a discrete-event network simulator. The concepts used in the development of UMJaNetSim were adopted from the NIST ATM/HFC Network Simulator (Section 2.2.2.1). UMJaNetSim was developed using the Java programming language and can be run under either the GUI or non-GUI mode.

UMJaNetSim allows the simulation of various network configurations and traffic loads. Researchers and network planners can use it to study the behavior of network protocols. A more detailed description of the structure and the components of UMJaNetSim is given in Chapters 4 and 5 respectively.

2.3 : Network Traffic Models

Network traffic models are used in traffic engineering to predict network performance and to evaluate congestion control schemes. Traffic models vary in their ability to model various correlation structures and marginal distributions (Adas, 1997). In network simulation, it is important to use traffic models that capture the statistical characteristics of actual traffic; models that are unable to do this result in poor network performance.

Subsection 2.3.1 lists the different types found in today's networks. The following subsection, subsection 2.3.2, gives an overview of the different models that can be used to simulate network traffic.

2.3.1 : Types of Network Traffic

There are different types of traffic in today's networks. Listed below are some types of traffic :

(i) VBR (Variable Bit Rate) traffic

(ii) Ethernet traffic

(iii) ATM (Asynchronous Transfer Mode)

(iv) WAN (Wide Area Network) traffic

(v) Telnet traffic

(vi) FTP (File Transfer Protocol) traffic

(vii) TCP data traffic

However, it is not easy to make a strict distinction between these traffic types, as transporting multimedia traffic will be a dominant factor in future networks (Hlavacs *et al*, 1999). For example, VBR encoded video will be transported over ATM and Ethernet networks, and multimedia traffic will be an important part of web traffic.

2.3.2 : Types of Network Traffic Models

While generating artificial network traffic, streams of requests can occur on several different levels of description. Stream *S* of a request is characterized by a sequence of observations

$$\ldots, X(t_{n-1}), X(t_n), X(t_{n-+1}), \ldots$$

at time points

$$\ldots, t_{n-1}, t_n, t_{n+1}, \ldots\ldots$$

The $X(t_i)$, are usually modeled by a family of random variables with a known probability distribution function and time index t. If the set of possible values (the state space) is finite/countable, the process is called a discrete-state process and if not, it is called a continuous-state process.

2.3.2.1 : Renewal Models

In a renewal process, the $X(t)$ are independent and identically distributed but their distribution function is allowed to be general. Independent here means that the observation at time t does not depend on any observation in the past or future, the autocorrelation function for all lags $k \neq 0$ is therefore equal to zero.

Renewal processes can be used to model arrivals that are strictly independent from each other. For example :

(i) The arrival of users to a company/computer facility

(ii) The arrival of network traffic packets, if the observed network traffic shows no autocorrelation

(iii) A stream of commands issued to an application, if no interdependencies on past results are observed.

Examples of renewal models :

(i) Poisson Processes

(ii) Bernoulli Processes

2.3.2.2 : Markov Models

Markov processes describe dependencies between the $X(t)$. A Markov process with discrete state space is called a Markov chain. A set of random variables $\{X_n\}$is called a discrete-time Markov chain if the probability that the next observed value (state) will be $x_{n+1} = j$ depends only on the current state $x_n = i$ and is given by p_{ij}. The dependency thus reaches back one unit in time and is also independent of the time the property has spent in its current state.

Markov processes can be used to model processes where the observations depend only on the previous observed value :

(i) User behavior : The next action is determined by the previous action including perhaps a return value

(ii) Network or system state change/failure

(iii) Network traffic, if the observed traffic shows none or little autocorrelation

Examples of Markov models :

(i) Markov Modulated Traffic models

(ii) Markov Modulated Poisson models

2.3.2.3 : Self-Similar Traffic Models

Empirical measurements of traffic have often shown the property of self-similarity. Self-similarity can be described by the Hurst parameter, H for which :

$$0.5 \le H \le 1$$

H= 0.5 indicates no self-similarity, H=1 indicates perfect self-similarity. If the equality holds only for variances and autocorrelation functions, the process is called second order self-similar.

Self-similar traffic has been observed in Ethernet and ATM traffic, Telnet and FTP traffic, web traffic and VBR video traffic. The topic of self-similarity regarding network traffic is discussed further in the next chapter (Chapter 3).

CHAPTER 3 : SELF-SIMILAR TRAFFIC

3.1 : Introduction

Studies on the nature of high-speed network traffic such as Ethernet LAN traffic (Leland *et al*, 1995) and VBR video traffic (Garrett & Willinger, 1994), (Beran *et al*, 1995) have shown that these types of traffic exhibit self-similar characteristics which are not captured by traditional traffic models such as Poisson and Markovian models.

The first section of this chapter provides a definition of the term 'self-similar'. The following section, Section 3.3 defines important terms that are closely related to self-similarity.

Section 3.4 discusses methods used to estimate the self-similarity of given data traffic and the succeeding section discusses two examples of self-similar data traffic; Ethernet traffic and VBR video traffic. The last section, Section 3.6, mentions a few methods to generate self-similar traffic.

3.2 : Self-similarity

The term 'self-similar' was first coined by Benoit B. Mandelbrot, a renowned mathematician. The concept of self-similarity is related to the concepts of chaos theory and fractals, and has been implemented in a variety of fields such as astronomy and mathematics.

An occurrence that is self-similar behaves the same way when looked at different degrees of resolution or different scales on a dimension. The Cantor set, which is widely used in books on fractals and chaos, helps illustrate the concept of self-similarity (Stallings, 1998). Figure 3.1 depicts the construction of a Cantor set with five levels of recursion. The following rules are applied:

(i) The set begins with the closed interval [0,1] which is represented by a line segment.

(ii) The middle third of the line is removed.

(iii) The middle third of the lines created by the preceding step is removed for each succeeding step.

The recursive process above can be also be defined as follows:

S_i represents the Cantor set after i levels of recursion.

$S_0 = [0,1]$

$S_1 = [0, 1/3] \cup [2/3,1]$

$S_2 = [0, 1/9] \cup [2/9, 1/3] \cup [2/3, 7/9] \cup [8/9, 1]$

$S_3 = \{ [0, 1/27] \cup [2/27, 1/9] \cup [2/9, 7/27] \cup [8/27, 1/3] \cup$

$[2/3, 19/27] \cup [20/27, 7/9] \cup [8/9, 25/27] \cup [26/27,1] \}$

and so on.

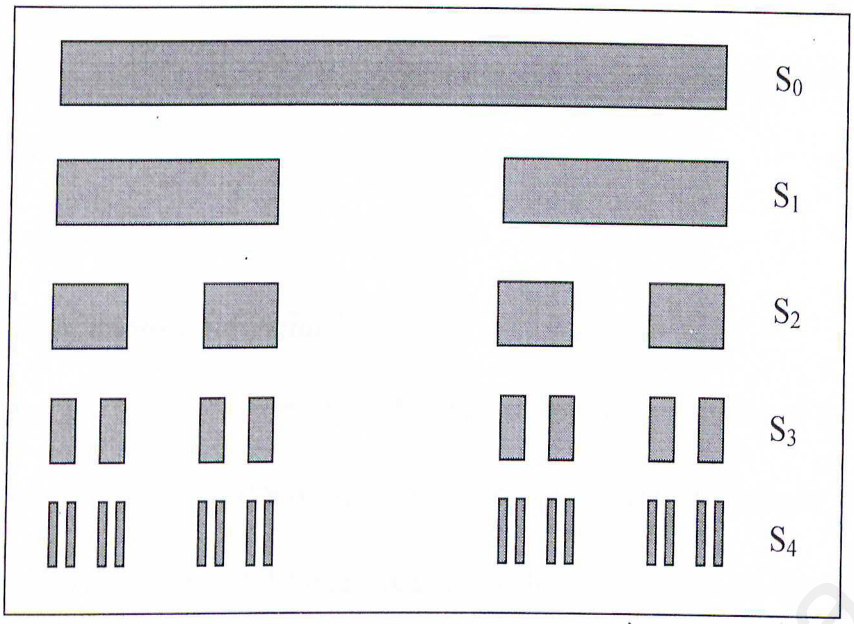


Figure 3.1 : Cantor set with five levels of recursion

It can be seen from the Cantor set that there are structures at small scales, and that these structures repeat. A self-similar structure contains smaller replicas of itself at all scales (Stallings, 1998). It is noted however, that these properties do not hold *ad infinitum* for real-life phenomenon, and do breakdown after a certain point.

3.3 : Self-Similar Data Traffic

Figure 3.1 gives a pattern that is reproduced exactly at different scales; it is consequently called exact self-similarity. Exact self-similarity can be constructed for a deterministic time series (Chow, 2001). It is better however, to view data traffic as a stochastic process.

There are two ways of defining stochastic processes: continuous-time definition and discrete-time definition, and these are mentioned in subsections 3.3.1 and 3.3.2

respectively. The remaining subsections define certain important terms related to self-similar data traffic such as the Hurst parameter, H, and long-range dependence (LRD).

3.3.1 : Continuous-time Definition

Continuous-time definition can be defined based on a direct scaling of the continuous time variable (Stallings, 1998). A stochastic process, $\mathbf{x}(t)$ is statistically similar with parameter H $(0.5 \le H \le 1)$ if for any real $a > 0$, the process $a^{-H}\mathbf{x}(at)$ has the same statistical properties as $\mathbf{x}(t)$. Parameter H, the Hurst parameter, is defined in subsection 3.3.3.

The following conditions can describe the relationship:

1. $\mathrm{E}[(\mathbf{x}(t)] = \dfrac{\mathrm{E}[\mathbf{x}(at)]}{a^{H}}$ Mean

2. $\mathrm{Var}[(\mathbf{x}(t)] = \dfrac{\mathrm{Var}[\mathbf{x}(at)]}{a^{2H}}$ Variance

3. $\mathrm{R}_{\mathbf{x}}(t,s) = \dfrac{\mathrm{R}_{\mathbf{x}}(at,as)}{a^{2H}}$ Autocorrelation

An example of a process that is based on this definition is the fractional Brownian motion (FBM) process. This process is a generalization of the more familiar Brownian motion process.

3.3.2 : Discrete-time Definition

For a stationary time series **x**, an *m*-aggregated time series $\mathbf{x}^{(m)} = \{\mathbf{x}_k^{(m)}, k = 0,1,2, ..\}$ can be defined by summing the original time series over adjacent and non-overlapping blocks of size *m*. This can be expressed as :

$$\mathbf{x}_k^{(m)} = \frac{1}{m} \sum_{i=km-(m-1)}^{km} \mathbf{x}_i$$

A process **x** is said to be exactly self-similar with parameter β (0< β < 1) if for all m = 1, 2, 3, .. we have (Stallings, 1998) :

$$\mathrm{Var}(\mathbf{x}^{(m)}) = \frac{\mathrm{Var}(\mathbf{x})}{m^{\beta}} \qquad \text{Variance}$$

$$\mathrm{R}_{x^{(m)}}(k) = \mathrm{R}_{\mathbf{x}}(k) \qquad \text{Autocorrelation}$$

Parameter β is related to the Hurst parameter, H, whereby :

$$H = 1 - (\beta/2)$$

For a stationary process, the variance of the time average decays to zero at the rate of 1 / *m*. For self-similar processes, the variance of the time average decays more slowly.

3.3.3 : Hurst Parameter, H

The Hurst parameter, *H*, is the parameter of self-similarity. It was named after H.E. Hurst, a hydrologist, who studied the water-flow behavior of different rivers.

The Hurst parameter represents the degree of self-similarity in the observed traffic. If the value of the Hurst parameter is between 0.5 and 1, the traffic is said to be self-similar (Ramakrishnan, 1999). The nearer H is to 1, the more self-similar the traffic, while a value of $H = 0.5$ indicates the absence of self-similarity. To be more precise, H is a measure of the persistence of a statistical phenomenon and is a measure of the length of the long-range dependence of stochastic processes (Stallings, 1998).

3.3.4 : Long-range Dependence

Long-range dependence is a very significant property of self-similar processes. It reflects the existence of clustering and bursty characteristics at all time scales – which are the persistent characteristics of self-similar processes.

Long-range dependence is defined in terms of the autocovariance C(τ) as τ increases. The autocovariance of many processes rapidly decays with τ. For example, a short-range dependent process satisfies the condition that its autocovariance decays at least as fast as exponentially (Stallings, 1998) :

$$C(k) \sim a^{|k|} \quad as \ |k| \to \infty, \quad 0 < a < 1$$

where ~ denotes that the two expressions on the two sides are asymptotically proportional to each other.

A long-range dependent process, on the other hand, exhibits a hyperbolically decaying autocovariance :

$$C(k) \sim |k|^{-\beta} \quad as \ |k| \to \infty, \quad 0 < \beta < 1$$

β is related to the Hurst parameter, H, where $H = 1 - (\beta/2)$.

3.3.5 :Heavy-tailed Distributions

Heavy-tailed distributions and long-range dependence are very closely related. A random variable Z has a heavy-tailed distribution if

$$\Pr\{Z > x\} \sim cx\text{-}\alpha, \qquad x \to \infty,$$

where $0 < \alpha < 2$ is called the tail index (or shape parameter), and c is a positive constant. The tail of the distribution decays hyperbolically (Willinger & Park, 2000).

An example of a heavy-tailed distribution is the Pareto distribution, which has been observed in various phenomena. The tail of the Pareto distribution decays much slower than exponential; and thus the term *heavy tail* (Stallings, 1998).

3.3.6 : Spectral Density

In the frequency domain, an equivalent formulation of long-range dependence can be stated. In more specific terms, the power spectral density obeys a power law near the origin:

$$S(\omega) \sim \frac{1}{|\omega|^{\gamma}} \quad \text{as } \omega \to 0,\ 0 < \gamma < 1$$

The spectral density for a discrete-time stochastic process is defined as :

$$S(\omega) = \sum_{k=-\infty}^{\infty} R(k)e^{-jk\omega} \qquad \gamma = 1 - \beta = 2H - 1$$

For short-range dependence (SRD) processes, the power spectral density remains finite as $\omega \to 0$.

3.4 : Self-Similar Estimation Techniques

This section discusses a few ways of testing for and estimating the degree of self-similarity of a given time series of data by estimating the Hurst parameter, *H*.

3.4.1 : R/S Plot

This is a graphical method of estimating the Hurst parameter and is based on the *rescaled adjusted range statistics R/S*, which was originally introduced by H.E. Hurst. For a stochastic process **x**(*t*) defined at discrete time instances $\{\mathbf{x}_t, t = 0, 1, 2, \ldots.\}$, the rescaled range of **x**(*t*) over a time interval N is defined as the ratio *R/S* :

$$\frac{R}{S} = \frac{\max\limits_{1 \le j \le N}\left[\sum_{k=1}^{j}(X_k - M(N))\right] - \min\limits_{1 \le j \le N}\left[\sum_{k=1}^{j}(X_k - M(N)\right]}{\sqrt{\frac{1}{N}\sum_{j=1}^{N}(X_k - M(N))^2}}.$$

where *M(N)* is the sample mean over the time period *N* :

$$M(N) = \frac{1}{N}\sum_{j=1}^{N} X_j$$

In the ratio above, the numerator is a measure of the range of the process. The denominator is the sample standard deviation. For a self-similar process, the ratio has the following characteristic for a large value of *N*:

$$R/S \sim (N/2)^H \qquad \text{with } H > 0.5$$

The equation above can be written again as :

$$\log[R/S] \sim H\log(N) - H\log(2)$$

If log $[R / S]$ versus N is plotted on a log-log graph, the result should fit a straight line with slope H.

3.4.2 : Variance-time Plot

The variance-time plot is another graphical method of estimating the Hurst parameter, H. For an aggregated time-series, $\mathbf{x}^{(m)}$ of a self-similar process, the variance is as follows (for a large value of m) :

$$\text{Var}\ (\mathbf{x}^{(m)}) \sim \frac{\text{Var}(\mathbf{x})}{m^{\beta}}$$

and the self-similarity parameter, $H = 1 - (\beta/2)$. The equation above can written as:

$$\log[\text{Var}\ (\mathbf{x}^{(m)})] \sim \log[\text{Var}(\mathbf{x})] - \beta\log(m)$$

log[Var($\mathbf{x}$)] is a constant that is independent of m, and therefore if Var ($\mathbf{x}^{(m)}$) versus m is plotted on a log-log graph, the result should be a straight line with a slope of $-\beta$. The plot can be generated easily from data series $\mathbf{x}(t)$ by generating the aggregate process at different levels of aggregation m and then computing the variance. Slope values between -1 and 0 suggest self-similarity.

3.4.3 : Whittle's Estimator

Whittle's estimator is a non-graphical method that provides confidence intervals. In this technique, it is assumed that the process is actually self-similar. It gives an estimate of the Hurst parameter with a certain confidence (Ramakrishnan, 1999). The autocorrelation and spectral density are defined as :

$$R(k) = E\,[\mathbf{x}(t)\mathbf{x}(t+k)] \qquad\qquad S(\omega) = \sum_{k=-\infty}^{\infty} R(k)e^{-jk\omega}$$

If it is assumed that the process is ergodic in correlation, then the autocorrelation can be estimated by:

$$\hat{R}_N(k) = \frac{1}{N}\sum_{n=0}^{N-1} X(n+k)X(n)$$

Since the spectral density S(ω) is the Fourier Transform of the autocorrelation function (R(k)), a Fourier operation on the estimate of the autocorrelation function will produce a good estimate of the spectral density.

The spectral density of a stochastic process $\mathbf{x}(t)$ defined at discrete time instances $\{\mathbf{x}_t,$ $t = 0, 1, 2, ..\}$ can be estimated by a Fourier series operation over a time period N, as shown below :

$$I_N(\omega) = \frac{1}{2\pi N}\left|\sum_{k=1}^{N} \mathbf{x}_k e^{jk\omega}\right|^2$$

This is known as the periodogram (or intensity function).

If an observed time series is assumed to be self-similar (with parameter H) and a particular form such as the fractional Brownian motion process is chosen, then the spectral density can be represented as $S(\omega, H)$. The value of the Hurst parameter, H is unknown but the form of the density is known. H can then be estimated by finding the value of H that minimizes the following expression:

$$\int_{\pi}^{\pi} \frac{I_N(\omega)}{S(\omega, H)} d\omega$$

This is known as the Whittle estimator.

This method can also estimate a sample variance so that confidence intervals can be computed. The sample variance:

$$\mathrm{Var}(\hat{H}) = 4\pi \left[\int_{\pi}^{\pi} \left(\frac{\partial \log S(\omega\omega)}{\partial H} \right)^2 d\omega \right]^{-1}$$

As mentioned previously, the Whittle estimator is different from the R/S plot and the variance-time plot as it assumes that the time series is a self-similar process.

3.5 : Examples of Self-Similar Data Traffic

Studies on the patterns of real-world data traffic have shown self-similar characteristics. This section discusses studies done on Ethernet traffic and VBR video traffic.

3.5.1 : Ethernet Traffic

The paper "On the Self-Similar Nature of Ethernet Traffic" (Leland *et al*, 1995) was a big breakthrough on the study of the behavior of Ethernet LAN traffic, as it showed that the traffic was self-similar and that traditional traffic models were unable to capture this characteristic. Rigorous statistical analysis was conducted on hundreds of millions of high quality Ethernet traffic measurements which were collected between1989 and 1992.

The main conclusions of the research:

(i) Ethernet LAN traffic is statistically self-similar, regardless of when and where the data during the four-year period was collected.

(ii) The degree of self-similarity, in terms of *H*, is a function of the overall utilization of the Ethernet and can be used to measure the "burstiness" of the data traffic.

(iii) Major components of the Ethernet LAN traffic(such as external LAN traffic) share the same self-similar characteristics as overall LAN traffic.

Figure 3.2 offers "pictorial proof" of self-similarity in Ethernet LAN traffic. The data displayed in the graphs was obtained from 27 consecutive hours of monitored Ethernet traffic (taken in August 1989). The features of the graphs in Figure 3.2 are listed below :

(i) Figure 3.2 depicts a sequence of simple plots of the packet counts (the number of packets per time unit) for five different choices of time units.

(ii) Beginning with a time unit of 100 seconds (graph (a)), each subsequent plot is obtained from the previous one by increasing the time resolution by a factor of 10, and by concentrating on a randomly chosen subinterval (as indicated by the darker shade). The time unit corresponding to the finest time scale is 10 milliseconds (graph (e)).

(iii) It can be observed that all the plots look very "similar" to one another in a distributional sense.

(iv) Looking at the scaling property (y-axis) and the absence of a natural length of a "burst" : it can be seen that **at every time** scale ranging from milliseconds to minutes to hours, **bursts consist of bursty subperiods followed by less bursty subperiods**.

The scale invariant (iv) or "self-simliar" feature of Ethernet traffic that can be observed from the graphs in Figure 3.2 is extremely different from traditional traffic models that have long been in use. The "pictorial proof" of the self-similar nature of Ethernet traffic motivates the use of self-similar stochastic processes for traffic modeling.

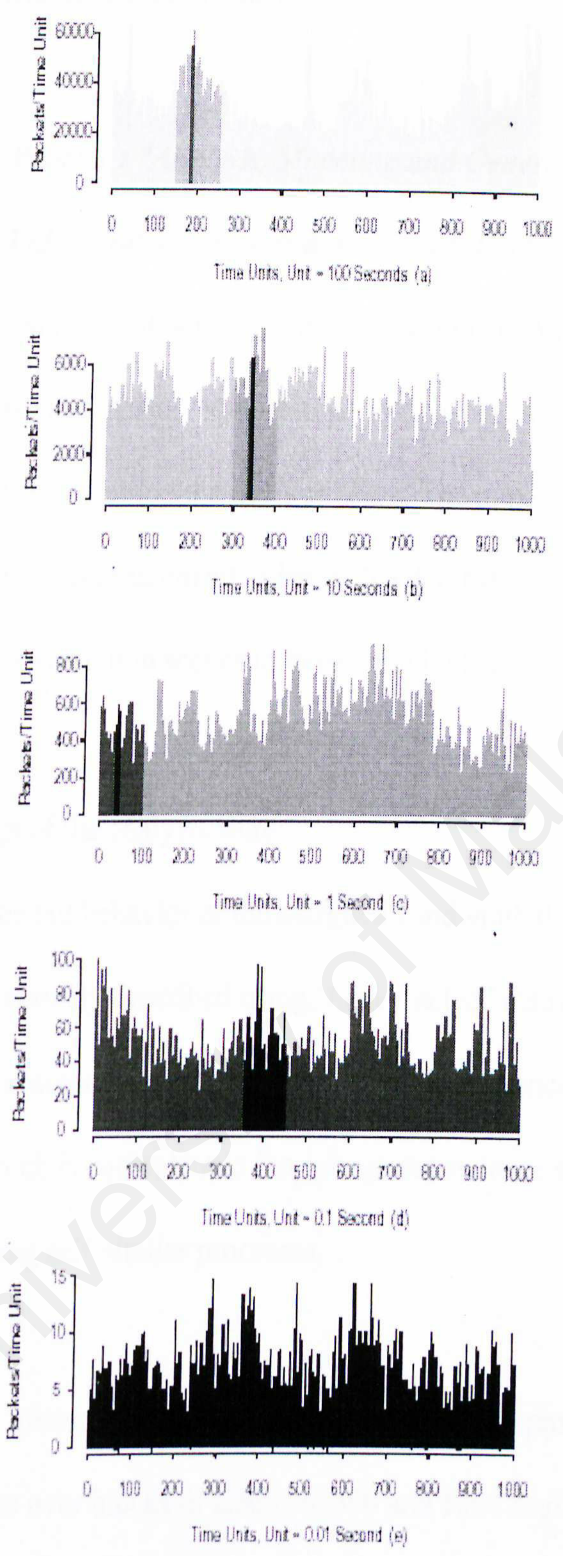


Figure 1 (a)—(e). Pictorial "proof" of self-similarity: Ethernet traffic (packets per time unit for the August '89 trace) on 5 different time scales. (Different gray levels are used to identify the same segments of traffic on the different time scales.)

Figure 3.2 : Pictorial "proof" of self-similarity in Ethernet traffic

3.5.2 : VBR (Variable Bit Rate) Video Traffic

3.5.2.1 : Paper 1 : "Analysis, Modeling and Generation of Self-Similar VBR Video Traffic" (Garett & Willinger, 1994)

A statistical analysis of a 2-hour long empirical sample of VBR video was conducted by the researchers. The sample was obtained by applying a simple intraframe video compression code to an action movie, *Star Wars*. The movie represents a realistic full-length sample of entertainment video with a diverse mixture of material ranging from low complexity/motion scenes to those with high action.

The main findings of the analysis were:

(i) The tail behavior of the marginal bandwidth distribution can be accurately described using "heavy tailed" distributions.

(ii) The autocorrelation of the VBR video sequence decays hyperbolically, which is equivalent to long-range dependence and can be modeled using self-similar processes.

Figure 3.3 is also "pictorial proof" of self-similarity. Three processes formed by aggregating frames over blocks of size 100, 500 and 1000 frames are compared. The graphs in Figure 3.3 not only retain significant correlations, but are similar in appearance.

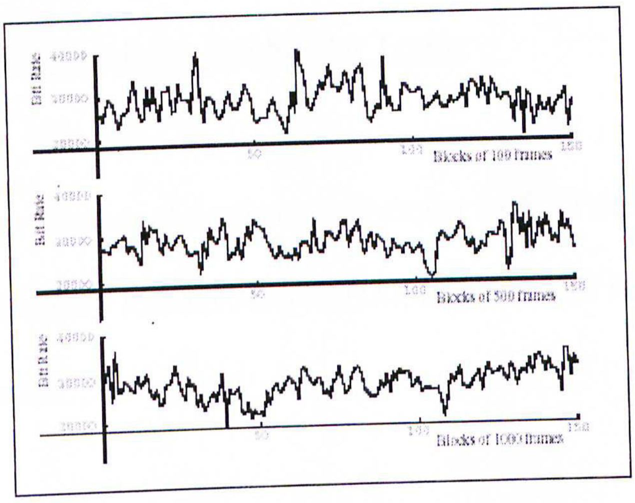


Figure 3.3 : Self-similarity in VBR video

3.5.2.2 : Paper 2 : "Long-Range Dependence in Variable–Bit-Rate Video Traffic" (Beran et al, 1995)

20 large sets of actual VBR video data were analyzed. These sets were generated by a variety of different codecs and they represented a wide range of different scenes. The main findings were :

(i) Long-range dependence is an inherent feature of VBR video traffic, and is a feature that is independent of scene (video phone, video conference or picture video).

(ii) The long-range dependence allowed the researchers to clearly distinguish between their measured data and traffic generated by the VBR source models that were currently being used.

3.6 : Methods of Simulating Self-Similar Traffic

There are many methods of simulating self-similar traffic. This section covers three such methods, namely Random Midpoint Displacement, Fast Fourier Transform and On/Off Processes.

3.6.1 : Random Midpoint Displacement (RMD)

The Random Midpoint Displacement (RMD) method can be used to create Fractional Brownian Motion (FBm). FBm can be used to model the sum or integral of self-similar traffic (as observed in network buffers, files sizes of video streams, etc). Its increments or derivatives can yield the self-similar fractional Gaussian noise (FGN).

The RMD method :

(i) Begin with two end-points

(ii) One point is added in the middle of these two points, and it is displaced with a random term (which depends on the Hurst parameter, *H*)

(iii) Points are added between all existing points and they are displaced with random terms, until the desired number of points has been generated.

3.6.2 : Fast Fourier Transformation (FFT)

This method of simulating self-similar traffic has shown to be fast and reliable. The strategy behind this method is to construct a sequence of complex values that correspond to the power spectrum of fractional Gaussian noise (FGN). The inverse discrete Fourier transform (IDTFT) can then be used to obtain the time-domain counterpart of this power spectrum. Since autocorrelation and power spectrum form a Fourier pair, it is guaranteed that the resulting process has the autocorrelation properties or self-similar properties of an FGN process.

3.6.3 : On/Off Processes

A large number of superimposed heavy tailed On/Off processes can produce self-similar traffic (Hlavacs *et al*, 1999). An On/Off process is either in state On or Off.

A time series can be constructed by observing the number of On-processes at any time point. If On-times and Off-times are drawn from a heavy tailed distribution with parameters α_1 and α_2, the observed stochastic process is a self-similar fractional Gaussian noise process with $H = (3 - \min(\alpha_1, \alpha_2))$.

On/Off processes can be used to create network traffic at the packet level, or streams of requests at a higher level (for example, transferring files over the network).

CHAPTER 4 : SYSTEM ANALYSIS

4.1 : Introduction

System analysis is a very crucial part of the development of any project. Without proper planning, installation of a system or module will lead to problems with implementation. The simulation of self-similar VBR traffic will be implemented as a module in the UMJaNetSim network simulator. It is therefore very important to analyze the existing structure of UMJaNetSim.

Section 4.2 gives an overview of the structure of the UMJaNetSim network simulator. The software requirements of the project are listed in Section 4.3. This section also includes the benefits of selecting the Java programming language and JCreator as the IDE (Integrated Development Environment) for developing the module. The last section lists the hardware requirements of the implementation.

4.2 : Overview UMJaNetSim Simulator

Listed below are the basic concepts of the UMJaNetSim network simulator:

(i) It is a discrete-event model

(ii) It has a central simulation engine with a centralized event manager

(iii) The simulation scenario is made up of a finite number of interconnected components (simulation object), each with a set of parameters (component properties)

(iv) Simulation execution involves components sending messages to one another. Messages are sent by scheduling an event which will occur at a later time for the target component

The concepts listed above are adopted from the NIST ATM/HFC Network Simulator. With this architecture, the simulator can simulate virtually "anything" that can be modeled by a network of components that send messages to one another.

Figure 4.1 is taken from the UMJaNetSim programmer guide. The simulation engine is the sole event manager and is responsible for the managing of all the user interface elements. It also provides convenient methods for file saving and data logging, as well as other tools. The programmer is responsible for the development of simulation components that directly represent the system that is to be simulated.

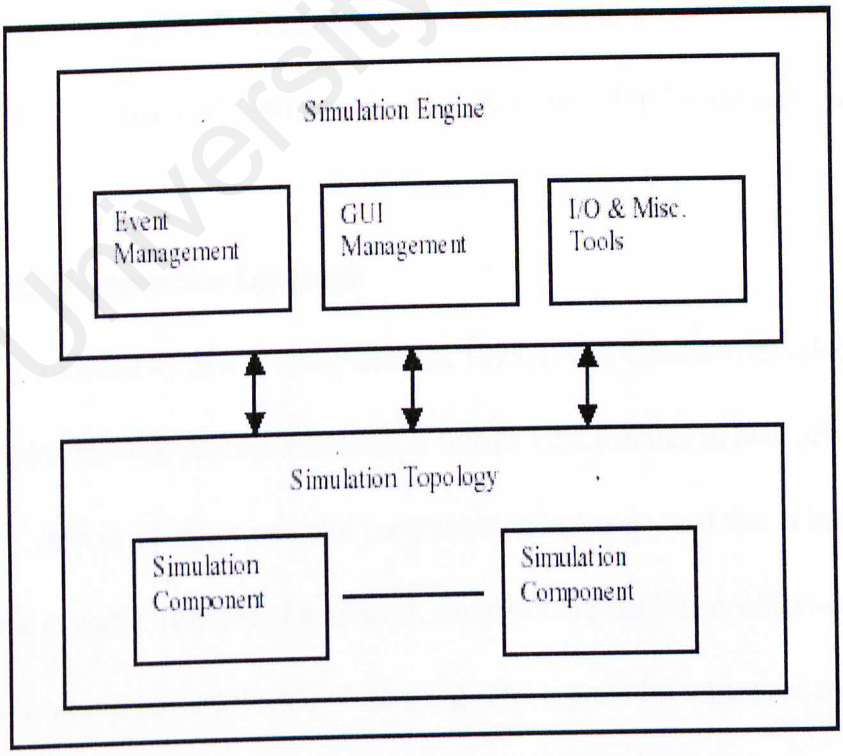


Figure 4.1 : Overall architecture of UMJaNetSim

4.3 : Software Requirements

The software listed below includes the programming language used as well the software used for the documentation of the project.

(i) *Microsoft Windows XP Professional Version 2002*: Operating System

(ii) *Java™ 2 Standard Edition Runtime Environment*: programming language

(iii) *JCreator 2.5LE*: IDE (Integrated Development Environment) used alongside the Java programming language

(iv) *BreezySwing and Terminal IO*: BreezySwing is a package of Java classes that simplifies the creation of a graphical user interface and the handling of interface events. It was used in the RS program (the RS program was used to test the self-similar data) to format the output

(v) *Microsoft Word 2000*: used for documentation

(vi) *Microsoft Project Professional 2002*: used for documentation

4.3.1 : Java Programming Language

Java was introduced by Sun Microsystems in 1995. It was chosen to develop the UMJaNetSim network simulator because it fulfills vital features of network simulators. Java is an object-oriented programming language, and this is beneficial because the essential features of a network simulator are built in an object-oriented approach. Java is a powerful and rich language and it provides important concepts of object-oriented programming such as class hierarchy, inheritance, polymorphism and encapsulation.

The programs created in Java are portable in a network. The source program is compiled into what Java calls bytecode, which can be run anywhere in a network on a server or client that has a Java virtual machine. The Java virtual machine interprets bytecode into code that will run on real computer hardware, which means that individual computer platform differences such as instruction lengths can be recognized and accommodated locally just as the program is being executed. This allows the network simulator to be run on any platform without doing any modifications to the program.

The Java programming language was expressly designed for use in the distributed environment system of the Internet. This allows the simulator to be used across the Internet by using a compatible Java browser.

Multithreaded operations are an important feature of network simulators. Java has built-in support for multithreading. This permits parallel operations of objects within the simulator.

Java code is also robust, which means that unlike programs written in C++, Java objects can contain no references to data external to themselves or other known objects. This characteristic ensures that an instruction cannot contain the address of data storage in another application or in the operating system itself, either of which would cause the program or even the operating system to terminate or "crash".

4.3.2 : JCreator 2.5LE

JCreator is a powerful IDE (Integrated Development Environment) for Java technologies. It provides the user with templates, Class browsers, a debugger interface, syntax highlighting, wizards and a fully customizable user interface.

Users can directly compile or run their Java programs without having to activate the main document first. JCreator will automatically find the file with the main method or the html file holding the applet,.and then start the appropriate tool.

Users can also create their own tools for calling Java Development Kit (JDK) applications such as the compiler, interpreter or applet viewer. JCreator also supports multiple compiler tools that can be switched with the runtime configuration dialogue box on the "Build" menu. Figure 4.2 shows a screenshot of JCreator 2.5LE.

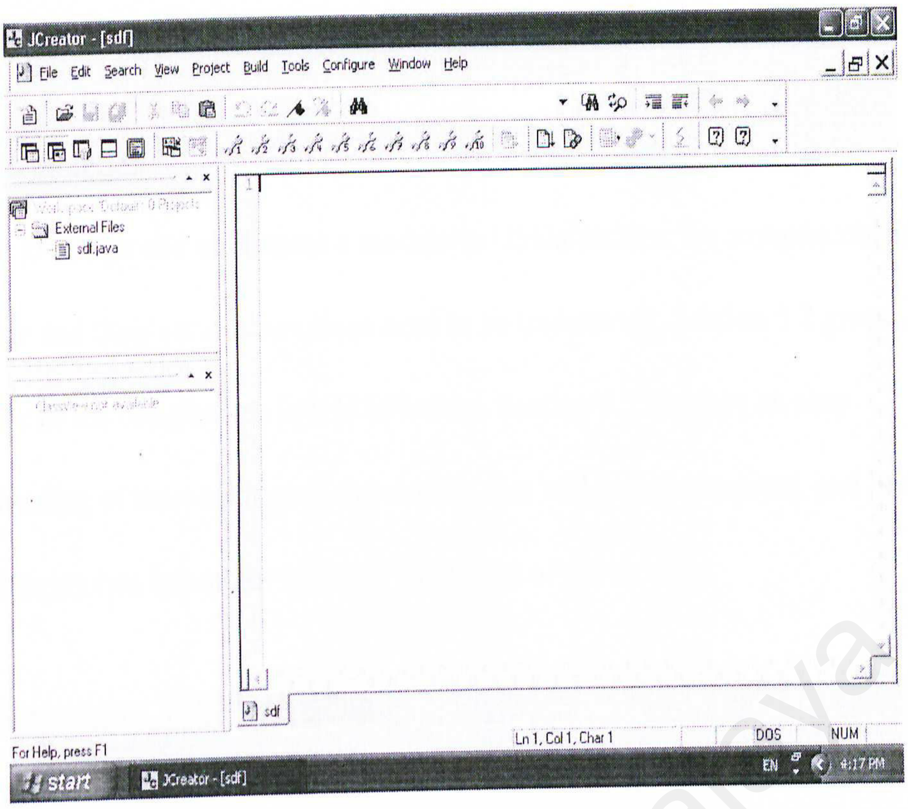


Figure 4.2 : Screenshot of JCreator 2.5LE

4.4 : Minimal Hardware Requirements

Listed below are the hardware requirements :

(i) Intel Pentium III processor

(ii) 733Mhz

(iii) 640MB of RAM

CHAPTER 5 : SYSTEM DESIGN

5.1 : Introduction

In order to design and implement a module in UMJaNetSim, the components of the simulator and their various functions need to be understood. Section 5.2 gives an overview of the components in UMJaNetSim. Section 5.3 displays an easy understanding of the workings of the module that will be implemented, and Section 5.4 highlights two important issues in the design of the module.

5.2 : Components of UMJaNetSim

In order to design and implement the VBR module in UMJaNetSim, it is vital to understand the existing components of UMJaNetSim. Figure 5.1 shows the components and their hierarchy in UMJaNetSim.

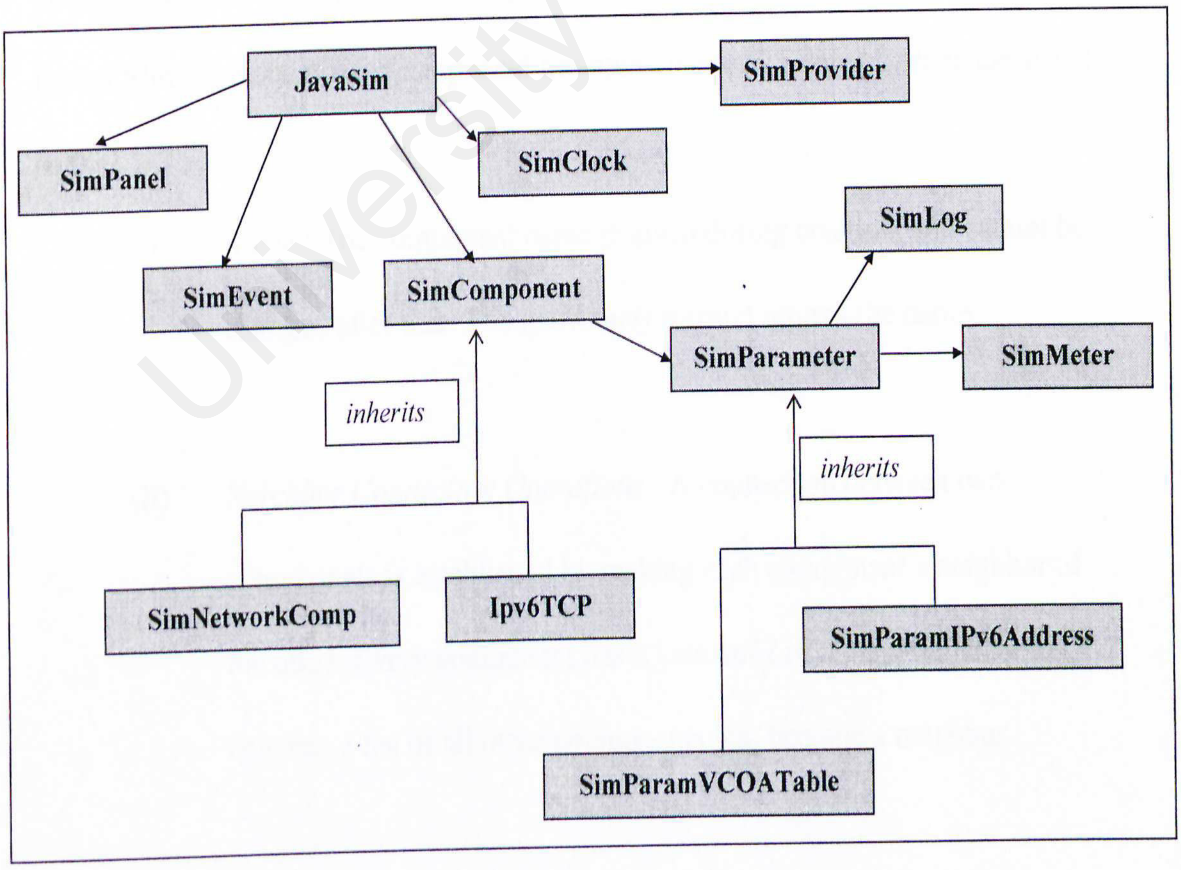


Figure 5.1 : Components in UMJaNetSim

5.2.1 : JavaSim

JavaSim is the main class in UMJaNetSim that contains everything in the simulator. It provides the event manager that handles event-passing among all components.

Some of the services provided by JavaSim are as follows :

(i) It provides the current simulation time in ticks

(ii) It provides a list of all existing SimComponent

(iii) It provides communication between components involved in the creation of a SimEvent

5.2.2 : SimComponent

This is the most important component in the simulator and it has to be understood in order to create new components. Every network component in the simulation MUST inherit SimComponent. SimComponent has many features, a few of which are listed below:

(i) *Name* : The component name is given during creation, and cannot be changed after that. The *getName()* method returns the name.

(ii) *Neighbor Connection Operations* : A connection between two components is established by making each component a neighbor of the other. Every component has a *java.util.List* named neighbors that contains a list of all other components that become a neighbor.

(iii) *Parameter List Handling* : Nearly every component has at least a few parameters which can be set by the user or can show something to the user such as the component status or simulation results. These parameters should be implemented as subclasses of the SimParameter (please refer to subsection 5.2.3) class in order to obtain features e.g. logging and on-the-fly graph-display.

The SimComponent component itself should not be instantiated as it only provides the skeleton for an actual component. A new component has to extend SimComponent and override its various methods in order to react to simulation events and other simulation operations.

5.2.3 : SimParameter

Each SimComponent can have internal parameters, which are neither shown nor accessible to users, or external parameters, which can be shown and are accessible to users. All external parameters, however, MUST inherit SimParameter. By extending SimParameter, logging and meter display features can be obtained automatically.

5.2.4 : SimClock

The simulation time of UMJaNetSim is based on "ticks". The duration of a tick is configurable in the simulator and by default a tick is equivalent to 10 nanoseconds. SimClock provides methods for the conversion between simulation ticks and real time.

5.2.5 : SimEvent

SimEvent defines simulation events. The SimComponent communicate with each other by enqueuing SimEvent objects for the target components.

To illustrate the use of SimEvent, the following example is used. If component A wants to send a packet to component B, component A creates a SimEvent that specifies B as its destination and enqueues the event. The SimEvent object contains a time so that this event is fired at exactly the specified time. Component B will then be able to react to the event.

There are two types of events; public (well-known) events and private events. A SimComponent can enqueue events for itself and another SimComponent. Private events can only be enqueued for itself, but public events can be enqueued for either itself or for another SimComponent. All private events are defined within the particular SimComponent source itself, whereas all public events are defined in the SimProvider object.

5.2.6 : Object Serialization and Load/Save Operations

UMJaNetSim uses object serialization as a form of light-weight persistence as this allows accurate saving and restoring of simulation states without much effort from the component developers. There are however, certain rules that need to be followed and they are briefly mentioned below :

(i) Every SimComponent and SimParameter must be Serializable

(ii) GUI (Graphic User Interface) members MUST NOT be serialized

(iii) Extra care must be taken when using *static* variables – as they are not serialized, their values are not saved.

5.3 : Simplified View of Implementation

This section looks at a simplified view of the implementation of the VBR module which generates self-similar traffic (Figure 5.2).

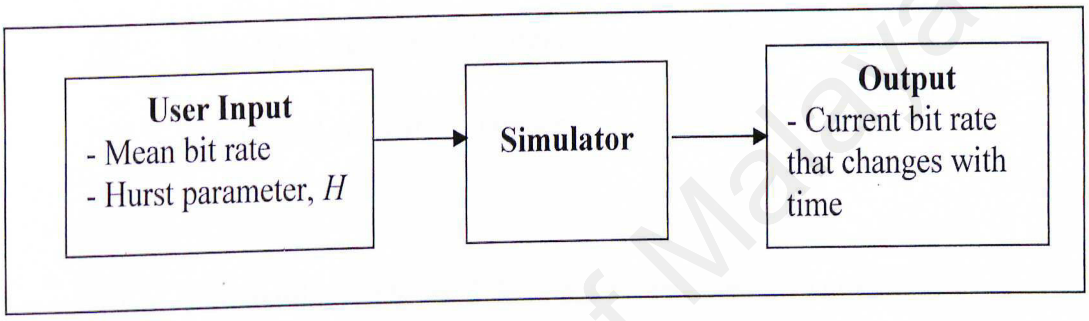


Figure 5.2 : Simplified View of Implementation

With the VBR module :

(i) The user inputs the mean bit rate and the Hurst parameter, *H*. As mentioned in Chapter 3 (3.3.3), the range of the Hurst parameter is : $0.5 < H < 1$

(ii) Using a self-similar simulation method (please refer to Chapter 3, section 3.6), the simulator uses the input and calculates the output which is the current bit rate that changes with time

The testing stage involves evaluating the output bit rate obtained to see whether or not it is self-similar (please refer to Chapter 3, section 3.4 for self-similar estimation techniques).

5.4 : Design Issues

Listed below are two important design issues in the design of the VBR traffic module:

(i) **User Interface**: The GUI interface is user-friendly ; it is clear and the words used are self-explanatory.

(ii) **Extensibility**: The module implemented is easy to modify and enhance. This will promote the improvisation of the module.

CHAPTER 6 : SYSTEM IMPLEMENTATION

6.1 : Introduction

The self-similar traffic source module was implemented in UMJaNetSim version 0.49 as an application component and this chapter discusses the implementation of the module. This chapter is divided into five main sections, the first being the introduction. Section 6.2 lists the steps taken in the implementation of the self-similar module. Section 6.3 contains a brief discussion on the method used to generate traffic, and Section 6.4 covers the implementation in UMJaNetSim 0.49 itself. Section 6.5 is a brief conclusion of this chapter.

6.2 : Implementation Steps

Listed below were the steps taken in the implementation of the self-similar module.

(i) The method of generating self-similar traffic was chosen – the Fast Gaussian Noise based on Vern Paxson's paper (Paxson, 1995)

(ii) The NIST ATM/HFC Simulator (written in C) had a traffic source module that generated self-similar traffic, and it was based on the paper written by Vern Paxson. The code for self-similarity was therefore in the simulator but it was comprised of many sub-programs. The sub-programs (written in C) were analyzed and compiled into a single standalone program.

(iii) The accuracy of the standalone C program in generating self-similar traffic was tested using the R/S statistic.

(iv) As the C program proved to be relatively reliable, it was translated to Java.

(v) The completed Java program was then tested for its accuracy using the R/S statistic.

(vi) Once the Java program was found to be capable of producing self-similar traffic, it was integrated into the UMJaNetSim (version 0.49) network simulator. A self-similar module was created in UMJaNetSim and the self-similar methods were integrated to it.

6.3 : Method of Generating Self-similar Traffic

The algorithm presented by Vern Paxson in his paper (Paxson, 1995) is based on synthesizing sample paths that have the same power spectrum as fractional Gaussian noise (FGN). The sample paths were then used in simulations as traces of self-similar network traffic. The algorithm is a fast approximation of the power spectrum of an FGN process; the approximation also has applications for fast estimation of the strength of long-range dependence (Hurst parameter) present in network arrival processes.

6.3.1 : Self-similar Traffic Module in the NIST ATM/HFC Network Simulator

The method used by Vern Paxson generated both positive and negative values. However, in the NIST ATM/HSC Network Simulator, a few changes were made to the original code so that only positive values were produced. This is because the values represent the bit rate which can only be positive.

In the NIST ATM/HSC Network Simulator, the FGN sample path X(k), with zero mean and variance σ^2, was scaled using the linear transformation :

$$A(k) = m + m*X(k) / (c* \sigma)$$

where:

m is the Mean Bit Rate specified the user

c is a scaling parameter set to 2 to ensure that 95% of the values of A(k) (which has a normal Gaussian distribution) will belong to the interval [0, 2*m]

After the transformation stated above, the negative values are set to zero and the values that are larger than 2*m are set to 2*m so that all the values of the new trace are within the interval [0,2*m]. Each rate of the trace is used to calculate the number of cell arrivals during one bin, whose duration is specified by the user.

6.4 : Implementation of VBR Self-Similar Module in UMJaNetSim

The VBR Self-similar traffic generator/traffic source module was implemented in UMJaNetSim version 0.49 as an application component (please refer to Figure 6.1).

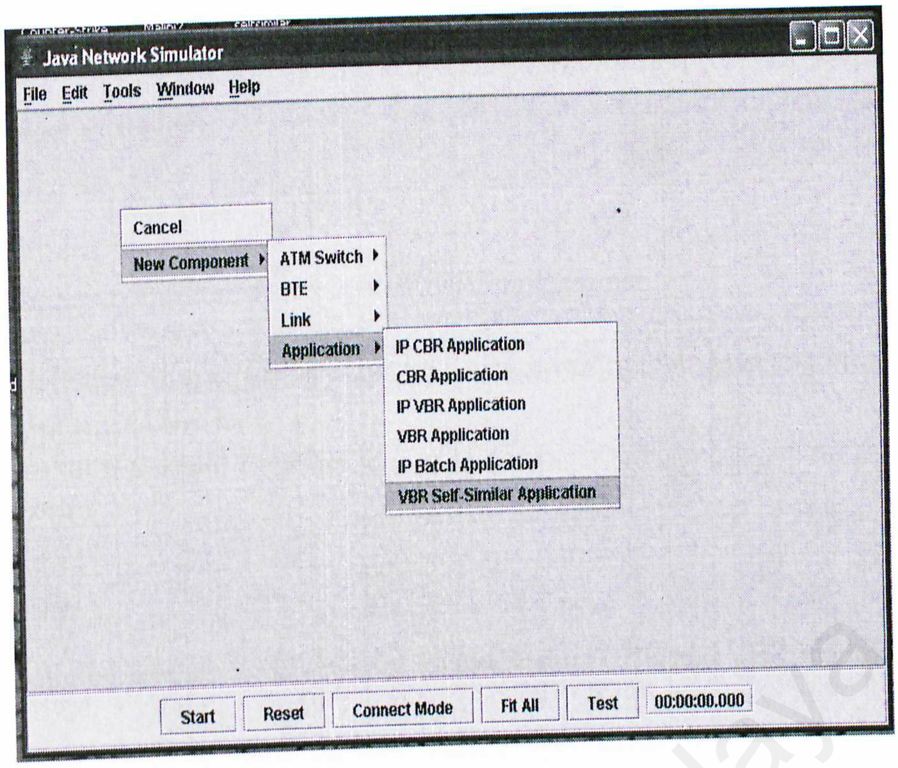


Figure 6.1 : VBR Self-Similar component in UMJaNetSim 0.49

The component name (in bold) was defined in the SimProvider and this is depicted in the following extract from SimProvider:

```
 private static final String [ ][ ] comps = {
   { "ATM Generic","ATM Switch","GenericATMSwitch" },
   { "ATM LSR", "ATM Switch", "ATMLSR" },
   { "BTE Generic", "BTE", "GenericBTE" },
   { "IP BTE", "BTE", "IPBTE" },
   { "Generic Link", "Link", "GenericLink" },
   { "IP CBR Application", "Application", "CBRIPApp" },
   { "CBR Application", "Application", "CBRApp" },
   { "IP VBR Application", "Application", "VBRIPApp" },
   { "VBR Application", "Application", "VBRApp" },
   { "IP Batch Application", "Application", "BatchIPApp" },
   { "VBR Self-Similar Application", "Application", "VBRSelfSimilar"}
//    { "TCP Application", "Application", "TCPApp" },
//    { "IP TCP Application", "Application", "TCPIPApp" }
  };
```

The component constant for the VBR Self-similar module was also defined in SimProvider as follows:

```
static final int VBRSS_APP = 12;
```

The important methods declared in VBRSelfSimilar were:

```
        VBRSelfSimilar(String name,int c,int t,JavaSim aSim,java.awt.Point loc) {
  super(name,c,t,aSim,loc);
  randgen=new java.util.Random();
  cn_create();
 }
```

The self-similar module is very similar to the existing CBR and VBR modules in the simulator and therefore shares the same basic properties such as:

(i) The neighbour operations:

```
boolean isConnectable (SimComponent comp);
void addNeighbor (SimComponent comp);
void removeNeighbor (SimComponent comp);
void removeNeighbors(java.util.List comps);
```

(ii) The Copy Operation :

```
void copy (SimComponent comp);
```

(iii) The initial/reset operations :

```
void reset();
void start();
void resume();
```

In order to provide a customized image for the VBRSelfSimilar component, the following method was invoked :

```
  Image getImage() {
  if(image==null) {
  //prepare the default image (name in a box)
    image=theSim.createImage(DEFAULT_WIDTH,DEFAULT_HEIGHT);
    Graphics g=image.getGraphics();
    g.setColor(Color.pink);
    g.fill3DRect(0,0,DEFAULT_WIDTH,DEFAULT_HEIGHT,true);
  }
  return image;
}
```

The color of the VBR Self-similar component was chosen to be pink.

6.4.1 : Topology Used in UMJaNetSim with the VBR Self-Similar Module as the Traffic Source

Figure 6.2 shows the topology used with the VBR Self-similar module as the traffic source. The VBR Self-similar module is connected to a BTE (Broadband Terminal Equipment) which in turn is connected to a link. The link is connected to an ATM Generic switch. Traffic generated at the module is received at the BTE on the 'other side' of the switch.

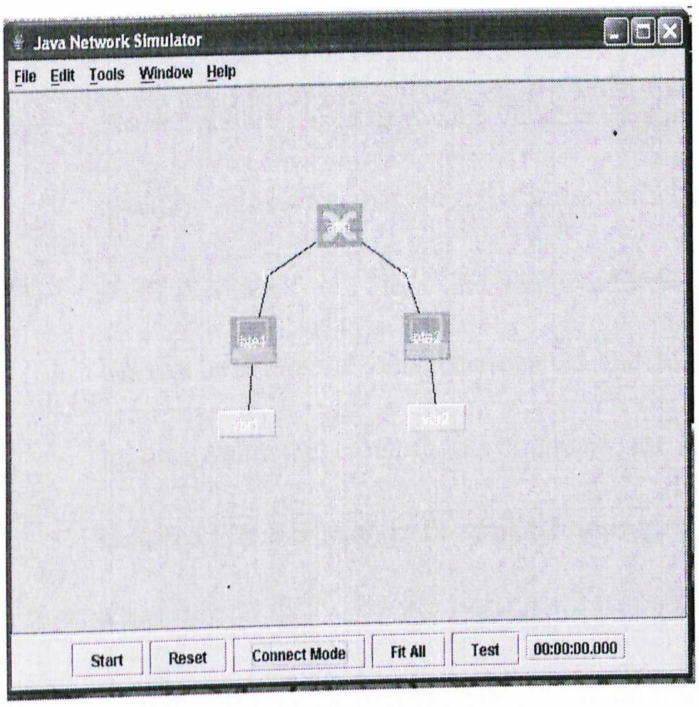
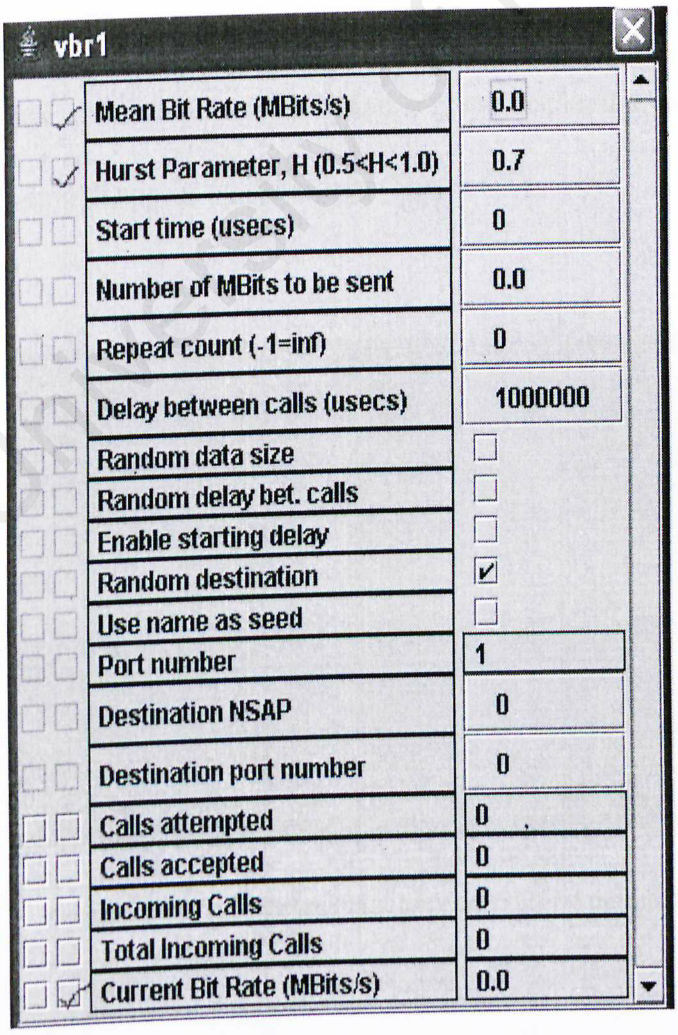


Figure 6.2 : Topology in UMJaNetSim with VBR Self-Similar module as traffic source

6.4.2 : Properties of VBR Self-Similar Module



vbr1

☐	✓	Mean Bit Rate (MBits/s)	0.0
☐	✓	Hurst Parameter, H (0.5<H<1.0)	0.7
☐	☐	Start time (usecs)	0
☐	☐	Number of MBits to be sent	0.0
☐	☐	Repeat count (-1=inf)	0
☐	☐	Delay between calls (usecs)	1000000
☐	☐	Random data size	☐
☐	☐	Random delay bet. calls	☐
☐	☐	Enable starting delay	☐
☐	☐	Random destination	✓
☐	☐	Use name as seed	☐
☐	☐	Port number	1
☐	☐	Destination NSAP	0
☐	☐	Destination port number	0
☐	☐	Calls attempted	0
☐	☐	Calls accepted	0
☐	☐	Incoming Calls	0
☐	☐	Total Incoming Calls	0
☐	✓	Current Bit Rate (MBits/s)	0.0

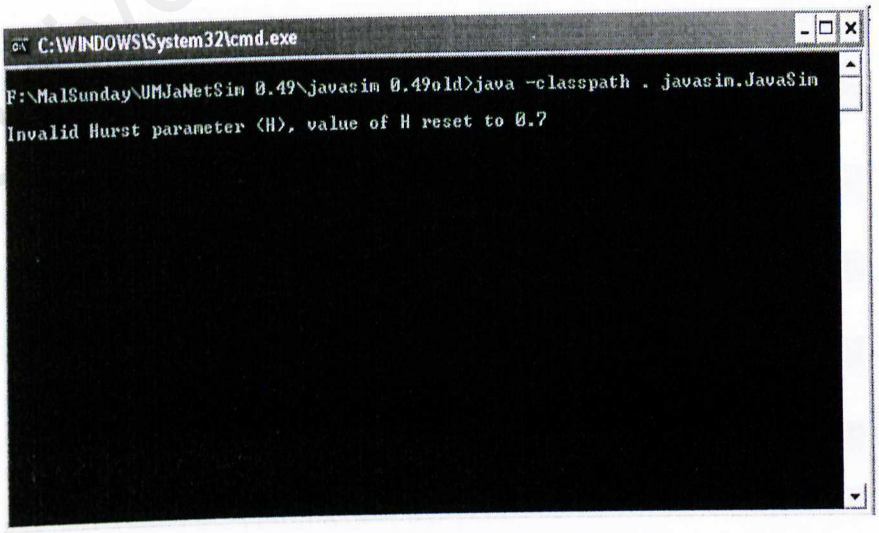
Figure 6.3 : VBR Self-Similar properties (unique properties ticked)

The self-similar module implemented has its own unique properties (Figure 6.3) :

(i) Mean Bit Rate (MBits/s) : This value is obtained from the user. The mean bit rate is used for the self-similar calculation.

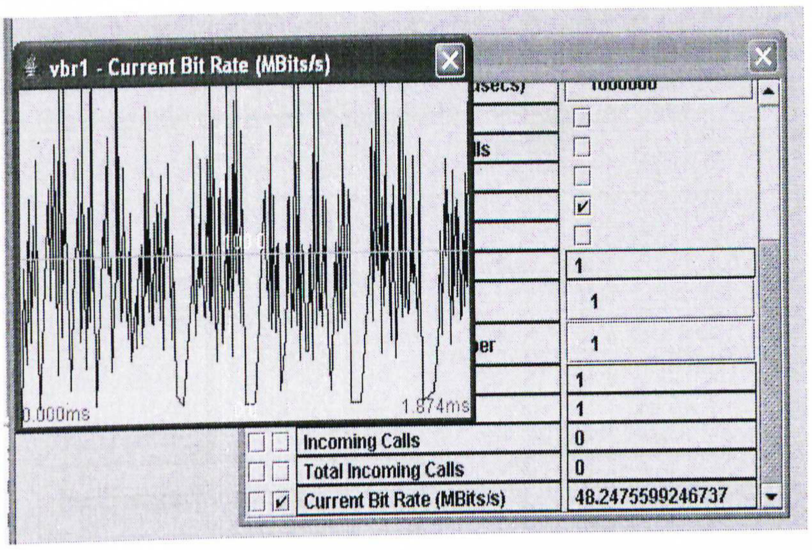
(ii) Hurst parameter, H : The Hurst parameter is initially defaulted to 0.7 but can be set to any value between 0.5 and 1.0. If the user however, inputs a value that is not in this boundary, the Hurst parameter is reverted to 0.7 with an error message appearing on the screen (Figure 6.4).

(iii) Current Bit Rate (MBits/s) : The current bit rate is the bit rate at a given time and it is constantly changing. The current bit rate is derived from the self-similar calculation and its value depends on both the Mean Bit Rate and the Hurst parameter. If the Current Bit Rate check box is ticked during simulation, a graph displaying the changing bit rate will appear on the screen (Figure 6.5).



```
F:\MalSunday\UMJaNetSim 0.49\javasim 0.49old>java -classpath . javasim.JavaSim
Invalid Hurst parameter (H), value of H reset to 0.7
```

Figure 6.4 : Error message that appears if Hurst parameter is not within 0.5 and 1.0



secs)	1000000
	☐
ls	☐
	☐
	✓
	☐
	1
	1
er	1
	1
	1
Incoming Calls	0
Total Incoming Calls	0
Current Bit Rate (MBits/s)	48.2475599246737

Figure 6.5 : Graph displaying the changing bit rate

6.4.3 : Self-Similar Calculation Methods

In the void start() method, the SelfSimilarGenerateFourier method is called with the following statement :

```
SelfSimilarGenerateFourier(12,hurst_parameter.getValue(), cn_bit_rate.getValue(),2);
```

The statement is explained as follows:

(i) 12 : this value determines the number of 'points' or self-similar values that will be generated. The number of values will be 4096 or 2^{12}. The number of points is defined as **n**

(ii) hurst_parameter.getValue() gets the value of the Hurst parameter

(iii) cn_bit_rate.getValue() gets the value of the Mean Bit Rate

(iv) 2 is the scaling factor. The scaling factor is used in the calculation and is always two (based on Vern Paxson's paper).

Figure 6.6 shows the methods involved in generating self-similar data.

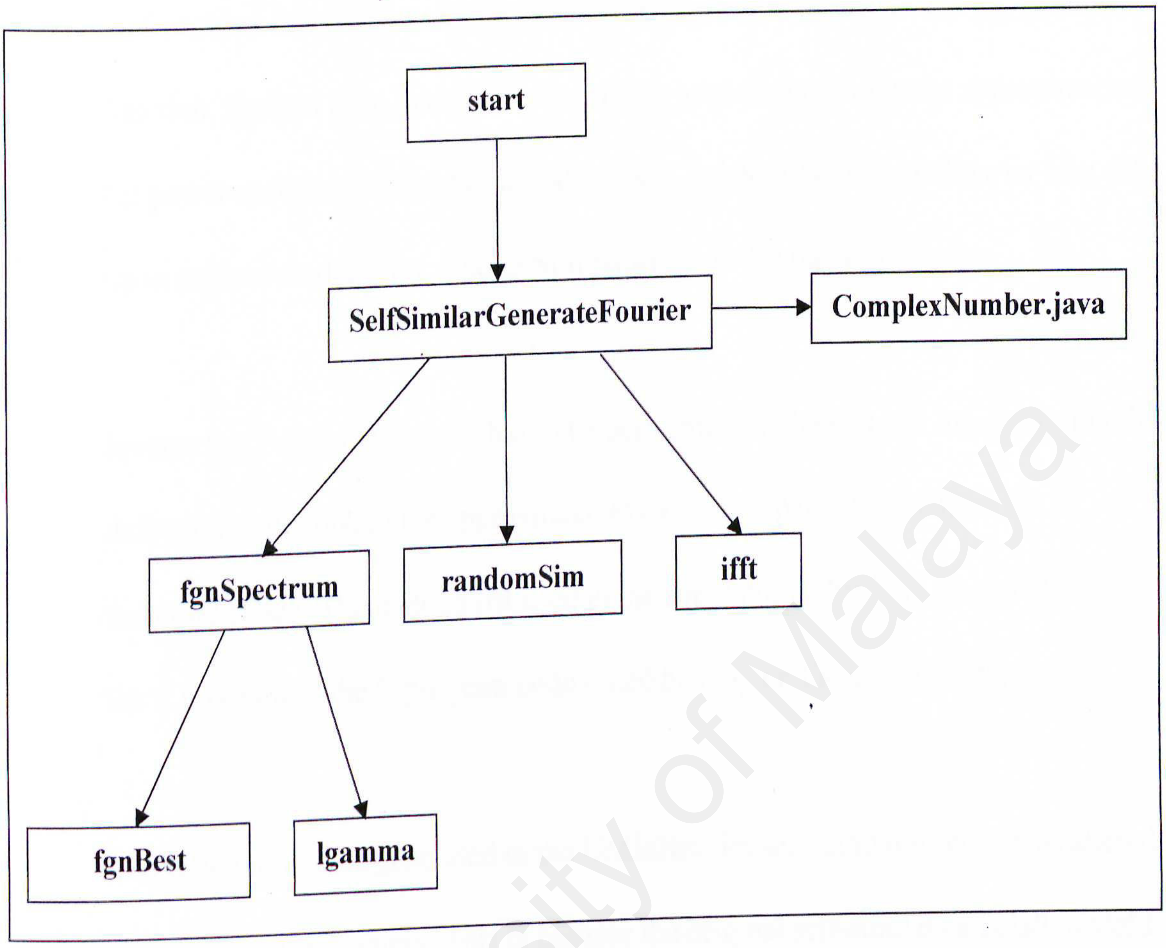


Figure 6.6 : Methods involved in generating self-similar data

In the calculation, complex numbers which consist of real numbers and imaginary numbers are used. The numbers are stored using a separate class called ComplexNumber.

The SelfSimilarGenerateFourier method calls the fgnSpectrum method. fgnSpectrum returns an approximation of the power spectrum for fractional Gaussian noise at the given frequencies lambda and Hurst parameter. Lambda is defined as :

$$\text{lambda} \leftarrow (i + 1) * \text{Pi} / \text{le}$$

where le = n/2
i = increases from 0 to le-1

fgnSpectrum calls upon methods fgnBest and lgamma. Lgamma is a log gamma function. fgnBest is an internal routine that is used in computing the approximation to the power spectrum. As part of its calculation, SelfSimilarGenerateFourier also calls upon method randomSim. randomSim returns a 31-bit random number.

Inverse fast Fourier transform has to be performed on the real and imaginary numbers defined in order obtain the approximate FGN sample path. The method ifft is therefore called. The method ifft used in the simulator is different from that used in the C program as the C program code relied heavily on the use of pointers.

The self-similar data generated in the UMJaNetSim self-similar module was altered slightly to suit the system. This is because the original self-similar calculation yields values that are equal to zero. The interval in sending cells in the simulator depends on the current bit rate (as shown in the statement below) :

```
long interval=SimClock.USec2Tick(424.0/currentBitRate[arrayCurrent]);
```

Should the currentBitRate at a given time be equal to zero, this would result in a system error as it involves a division with zero. The value of the current bit rate is therefore set to 1. As this would effect the 'self-similarity' of the data, the testing (Chapter 7) for self-similarity is conducted on the results of the standalone Java program instead of the results yielded by the simulator.

As mentioned previously, the self-similar calculation only provides values for 4096 points, and if large amounts of data need to be sent, 4096 values will be insufficient. It is for this reason that the self-similar values that are generated and stored in the array currentBitRate (which represents the current bit rate) are reused once the 4096th array has been reached.

6.5 : Conclusion

The most difficult part of the implementation process of this project was the conversion of the C code to Java. The C program relied heavily on the use of pointers, and this made the translation more challenging. As mentioned in the previous section, a few changes were made to the self-similar calculation that was implemented in the UMJaNetSim simulator and it is for this reason testing for the accuracy of self-similarity was conducted on the standalone Java program and not the simulator itself.

CHAPTER 7 : TESTING

7.1 : Introduction

This chapter is divided into four sections. The first part is an introduction. Section 7.2 discusses the method of testing. Section 7.3 lists the test results and the last section, section 7.4, lists the conclusions made based on the test results.

7.2 : Testing with the R/S Statistic

The R/S statistic was chosen as the method of testing to test the self-similar program in C as well the standalone program written in Java. The R/S test program, in Java, was written by Nithyanandan Natchimuthu and is based on concepts in a book entitled "Statistics for Long-Memory Processes" by Jan Beran (Beran, 1994).

The output values which are to be tested have to be written to a text file. The text file has to be in the same location as the R/S program. The R/S program is compiled with the following statement:

```
javac RS.java
```

Once the compilation is successful, the R/S program is executed with the following statement:

```
java RS {filename} {t_multiple} {k_multiple} {minData}
{maxData} {t_max} {k_mx} optional{}
```

For the testing, the following values were used :

```
java RS output.txt 200 50 0 0 0 0 1
```

(i) "output.txt" is the name of the text file which stores the values that are to be tested for self-similarity

(ii) t_multiple is the difference between the number of times the main loop in the code is run and the total number of points. The smaller the value of t_multiple, the more 'detailed' the estimation

(iii) k_multiple. 50 is the value used in testing. This means that estimation occurs every 50 points. As the total number of points is 4096, 50 is a suitable number.

(iv) minData is the point on the list of values where the estimation begins. A value of 0 means that the estimation should begin with the first point

(v) maxData is the point on the list of values where the estimation ends. A value of 0 means that the estimation will end on the last point of the list

(vi) t_max = 0 gives the full R/S statistic, anything else would truncate at the given t_max

(vii) Optional{0} gives only the Hurst parameter, any other integer gives the R/S statistic and the Hurst parameter.

7.3 : Testing Results

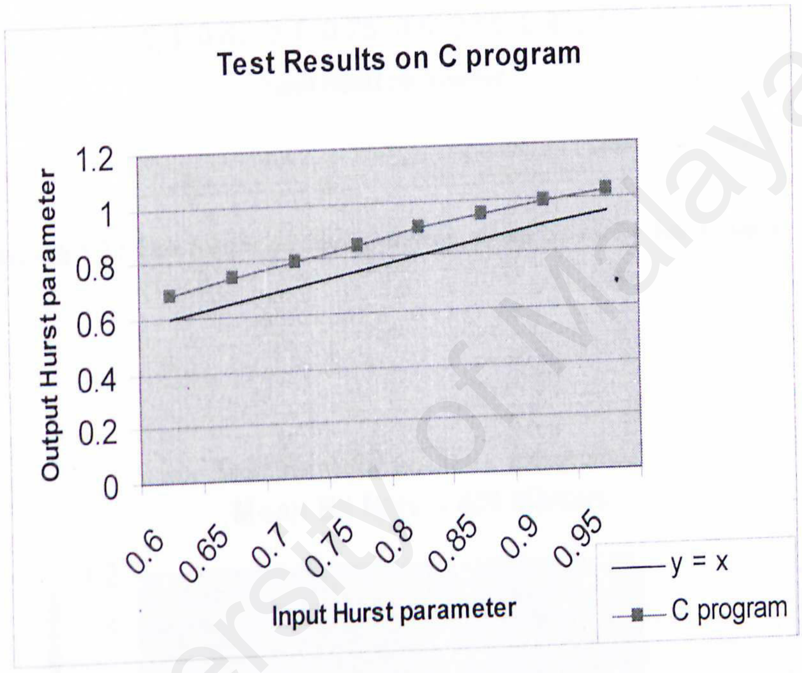
Testing with the R/S program was conducted on both the original C program and the standalone Java program. As mentioned in the previous chapter, testing with the R/S statistic was done on the standalone Java program and not the simulator as the simulator's self-similar calculation was slightly altered, and this would have had an effect on the results.

Testing was conducted with varying values of the Hurst parameter (nine values), ranging from 0.55 to 0.95 and with varying mean bit rates (100 MBits/s, 500 MBits/s, 1000 MBits/s and 2500 MBits/s). The values of t_multiple and k_multiple were fixed to 200 and 50 respectively. Table 7.1 displays the results :

Input Hurst parameter	**Output Hurst parameter**							
	100 MBits/s		**500 MBits/s**		**1000 MBits/s**		**2500 MBits/s**	
	Java	**C**	**Java**	**C**	**Java**	**C**	**Java**	**C**
0.55	0.424	0.631	0.565	0.631	0.602	0.631	0.634	0.631
0.60	0.614	0.686	0.595	0.686	0.642	0.686	0.674	0.686
0.65	0.703	0.741	0.689	0.741	0.639	0.741	0.657	0.741
0.70	0.748	0.795	0.780	0.795	0.719	0.795	0.786	0.795
0.75	0.783	0.849	0.796	0.849	0.751	0.849	0.780	0.849
0.80	0.812	0.900	0.768	0.900	0.850	0.900	0.788	0.900
0.85	0.851	0.946	0.934	0.946	0.846	0.946	0.923	0.946
0.90	0.884	0.986	0.988	0.986	0.869	0.986	0.898	0.986
0.95	0.908	1.019	0.984	1.019	0.870	1.019	0.975	1.019

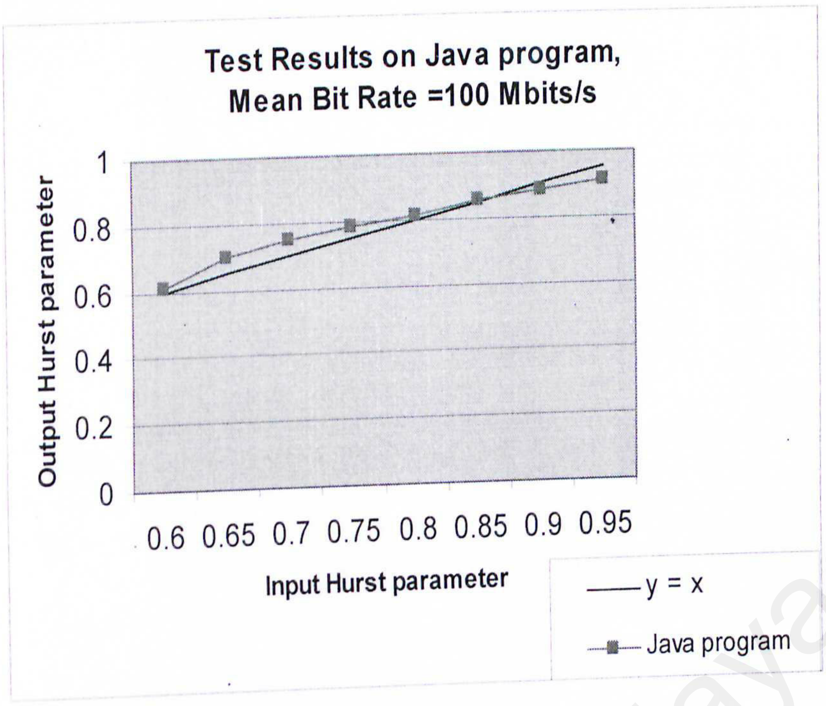
Figure 7.1 : Test results

Based on the table, it can be concluded that the output Hurst parameter for the C program is exactly the same at a given input Hurst parameter at varying mean bit rates. For example, at the given input Hurst parameter of 0.60, the output Hurst parameter for the C program is 0.686 regardless of the mean bit rate. Put in another way, the output Hurst parameter is only dependent on the Hurst input and not the mean bit rate. Consequently, only one graph is sufficient to display the results. Graph 7.1 displays the results of the testing on the C program outputs. The graph y = x is also drawn for comparison's sake.

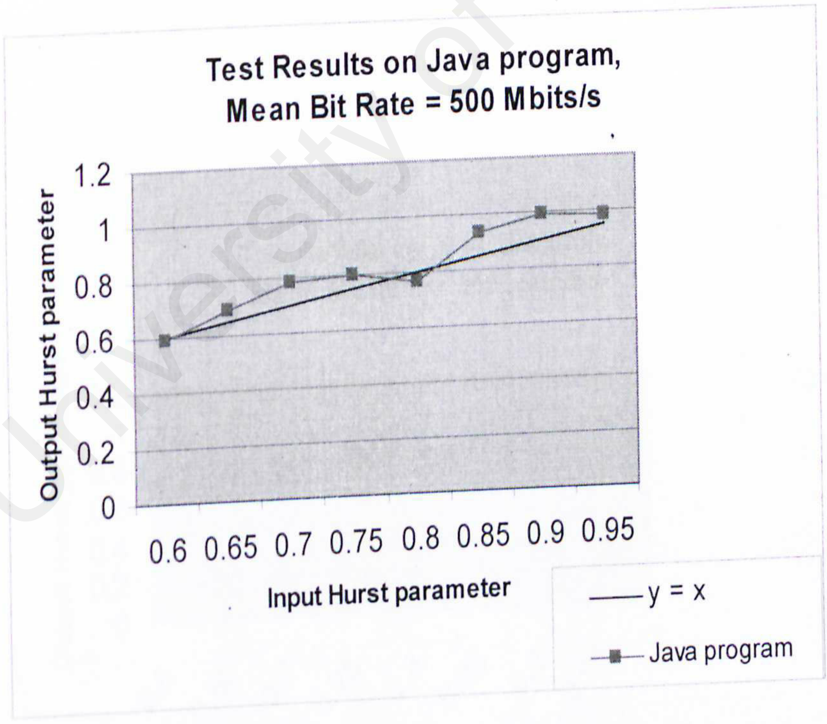


Graph 7.1 : Test results on C program

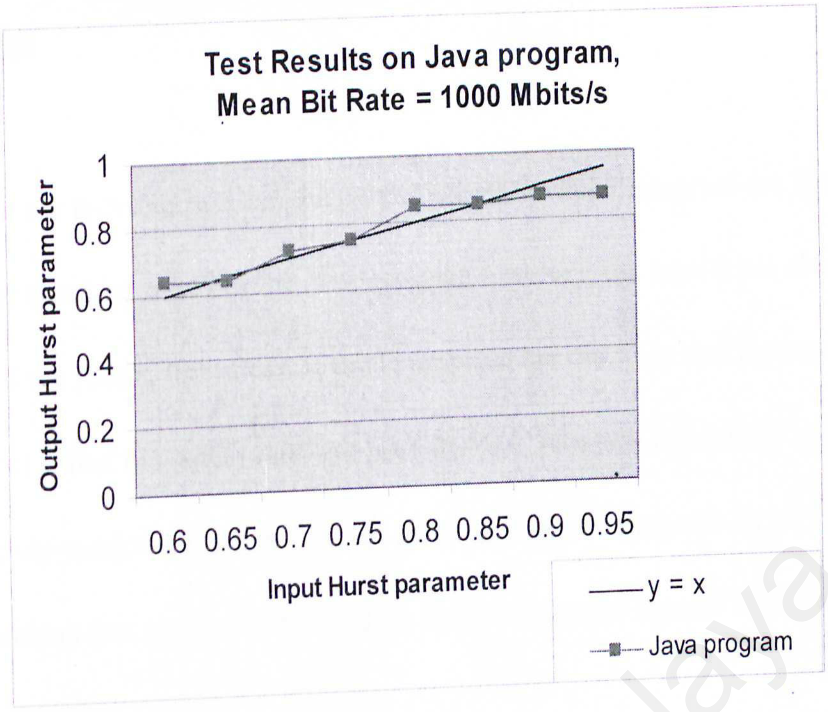
The following graphs display the results of the testing on the Java program at varying mean bit rates. Each graph contains the function "y = x" for the sake of comparison.



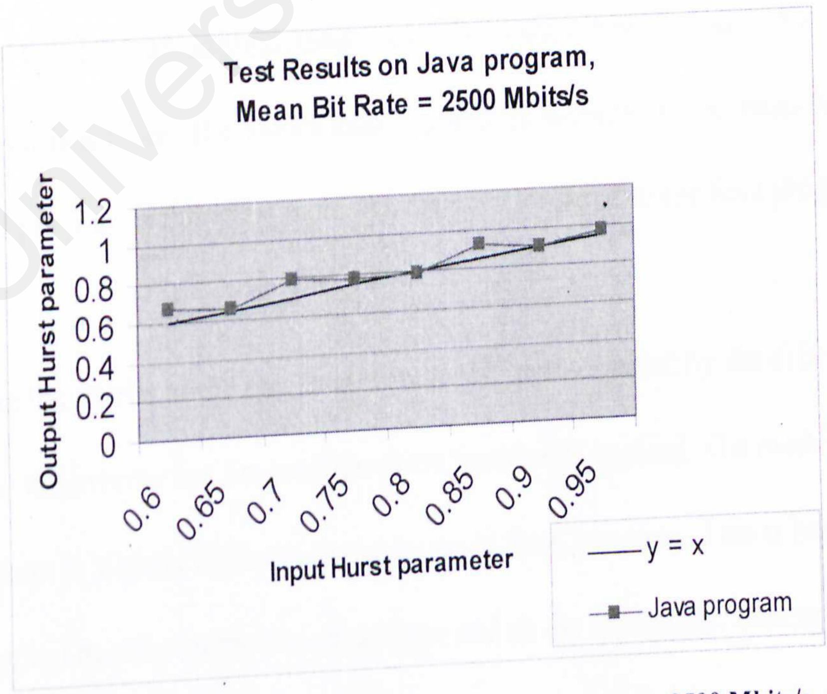
Graph 7.2 : Test results on Java program, mean bit rate = 100 Mbits/s



Graph 7.3 : Test results on Java program, mean bit rate = 500 Mbits/s



Graph 7.4 : Test results on Java program, mean bit rate = 1000 Mbits/s



Graph 7.5 : Test results on Java program, mean bit rate = 2500 Mbits/s

7.4 : Conclusions

As mentioned in the previous section, the output values for the C program are the same regardless of the mean bit rate. For the Java program however, the results are always different. Part of the reason the values of the C program are the same and the ones for are Java are different is that the rand() function used in the C program is not truly random as it generates pseudo-random numbers. Calling the rand() function repeatedly produces a sequence of numbers that appears to be random but the sequence repeats itself each time the program is executed.

While the Java output values are usually closer to the input value compared to the C output values, the C output values are more 'uniformed'. The output values for the C program are bigger than the input values, but they increase linearly. The output values for the Java program however are unpredictable : sometimes the output is bigger than the input, sometimes it is smaller. The values tend to get more accurate as the mean bit rate increases. The C program is therefore more predictable compared to the Java program.

The difference in the results of the two programs is also partly caused by the different method in which the inverse fast Fourier transform function is applied. The method used in the Java program is slightly different than the one in the C program. This is because the C program relied heavily on the use of pointers and as the Java programming language does not support the use of pointers, there could be no 'direct translation' of code.

Based on the results, it can be concluded that the Java program can produce self-similar traffic, although not with 100% accuracy. It has to be also noted that the R/S statistic is a heuristic approach and the results are an estimation and not completely accurate.

CHAPTER 8 : CONCLUSION

8.1 : Conclusion

The VBR Self-similar module is able to function in UMJaNetSim 0.49, eventhough the traffic generated is not 100% self-similar. This last chapter discusses the system's strengths and weaknesses, as well as possible enhancements that can be made in the future.

8.2 : System's Strengths

The FGN method of generating self-similar data is a fast method. The self-similar calculation code was integrated directly into the VBR Self-similar module, and this makes it easier to implement it in future versions of UMJaNetSim.

The "Current Bit Rate" property in the VBR Self-similar module allows users to view the changing values and therefore see firsthand the self-similar nature of the data generated.

Running the module is also easy as there is only an additional parameter to be set (the Hurst parameter).

8.3 : System's Weaknesses

The self-similar module has a few weaknesses. As the R/S statistic is a heuristic method of testing, it can sometimes be unreliable

As mentioned in previous chapters, the self-similar calculation yields values that are equal to zero. These zero values cannot be used in the simulator as it would lead to a division with zero. The zero values were consequently set to 1. This 'resetting' of values leads to an inaccuracy in the self-similarity of the data.

The Java program is not 100% accurate. The test results were rather unpredictable. Also, as mentioned in Section 6.3.1., the values of the self-similar data were 'bounded' so that they were between 0 and 2*m (mean bit rate). Setting boundaries on the data does have an effect on its self-similarity.

8.4 : Future Enhancements

Listed below are future enhancements that can be made to the module :

(i) Overcome the 'zero divisional' problem so that the self-similarity of data generated in the simulator is more accurate.

(ii) Incorporate the R/S program or another method of testing into the simulator itself so the accuracy of self-similarity can be viewed while running the simulator

(iii) Integrate the self-similar VBR module into the latest version of UMJaNetSim; UMJaNetSim version 0.66

(iv) Disallow the user from entering an invalid Hurst parameter as soon as the invalid parameter in entered instead of until the simulator starts running

REFERENCES

Books

1. STALLINGS, W., 1998. High-Speed Networks: *TCP/IP and ATM Design Principles*. New Jersey: Prentice-Hall International, Inc.

2. DEITEL, H.M. AND DEITEL, P.J., 2003. *Java How To Program*. 5th ed. New Jersey: Pearson Education, Inc.

3. KELTON, W. D. AND LAW, A.M., 2000. *Simulation Modelling and Analysis*. 3rd ed. McGraw-Hill International Series.

4. HEYMAN, D.P. AND LAKSHMAN, T.V., 2000. Long-Range Dependence and Queuing Effects for VBR Video. *In* : PARK, K. AND WILLINGER, W., eds. *Self-Similar Network Traffic and Performance Evaluation*. United States of America : John Wiley & Sons, Inc., 285-318.

5. CORNER, M., GOLMIE, N., KOENING, A., SAINTILLAN, Y. AND SU, D., 1998. *The NIST ATM/HFC Network Simulator*. U.S. Department of Commerce,(NISTIR 5703R1).

6. BERAN, J., 1994. *Statistics for Long-Memory Processes*. Chapman & Hall.

Theses

1. CHOW, C.O., 2000. *Traffic Prediction and ABR Congestion Control for ATM Networks Using Artificial Neural Network*. Thesis (Master). University of Malaya.

2. TAN, K.H., 2001. *Fuzzy APPD in ATM Network*. Thesis (Undergraduate). University of Malaya.

Articles/Journals

1. HLAVACS, H., KOTSIS, G. AND STEINKELLNER, C., 1999. *Traffic Source Modelling*. University of Vienna.Technical Report No. TR-99101.

2. ALBERTI, A., NETO, A., KLEIN, J. AND MENDES S.M., 1998. *SimATM: An ATM Network Simulation Environment*. Brazil. (DECOM-FEEC/UNICAMP)

3. RAMAKRISHNAN, P., 1999. *Self-Similar Traffic Models*. Center for Satellite and Hybrid Communication Networks, CSHCN T.R. 99-5.

4. ADAS A., 1997. Traffic Models in Broadband Networks. *IEEE Communications Magazine*, July 1997, 82-89.

5. LELAND, W.E., TAQQU, M.S., WILLINGER, W. AND WILSON, D.V., 1995. On the Self-Similar Nature of Ethernet Traffic. *Computer Communication Review,* 203- 213. Available from : http//www.acm.org/sigcomm/ccr.archive/1995/jan95/ccr-9501-leland.pdf [Accessed 9 August 2003].

6. GARETT, M.W. AND WILLINGER, W. Analysis, Modeling and Generation of Self-Similar VBR Video Traffic. *ACMSigComm*, September 1994.

7. BERAN, J., SHERMAN, R., TAQQU, M.S. AND WILLINGER, W. Long Range Dependence in Variable-Bit-Rate Video Traffic. *IEEE Transactions on Communications, Vol. 43, No. 2/3/4*, February, March, April, 1995, 1566-1579.

8. PAXON, V. *Fast, Approximate Synthesis of Fractional Gaussian Noise for Generating Self-Similar Network Traffic*. Laurence Berkeley Laboratory, Berkeley, LBL-36750, April 1995.

Other

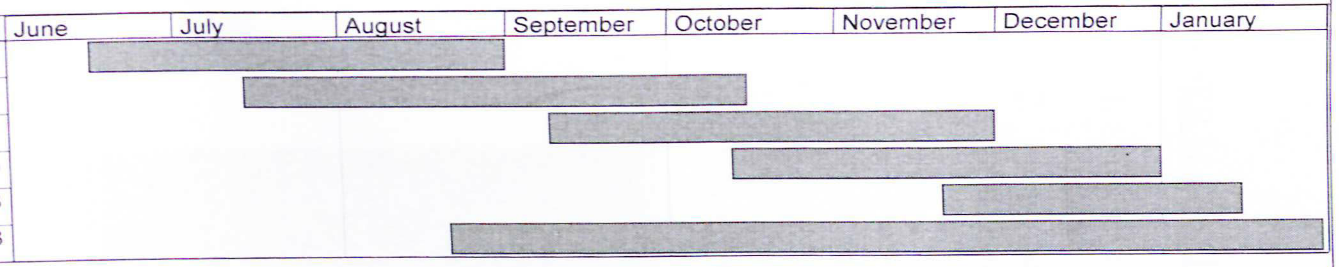
1. *JaNetSim 0.63 : Programmer's Guide*. March 2002, Makmal Sistem Rangkaian, University of Malaya.

APPENDIX

The first part of the Appendix consists of the Gantt chart for the project schedule (this is on the following page, Table A1).

The second part of the Appendix consists of the User Manual which is a guide to set up a simple topology in UMJaNetSim 0.49 with the VBR Self-Similar module as the traffic source.

ID	Task Name	Duration	Start
1	Literature Review	56 days	Mon 6/16/03
2	System Analysis	69 days	Mon 7/14/03
3	System Design	61 days	Tue 9/9/03
4	System Coding / Implementation	60 days	Mon 10/13/03
5	System Testing / Evaluation	42 days	Fri 11/21/03
6	Documentation	121 days	Thu 8/21/03



Project: ps
Date: Thu 2/19/04

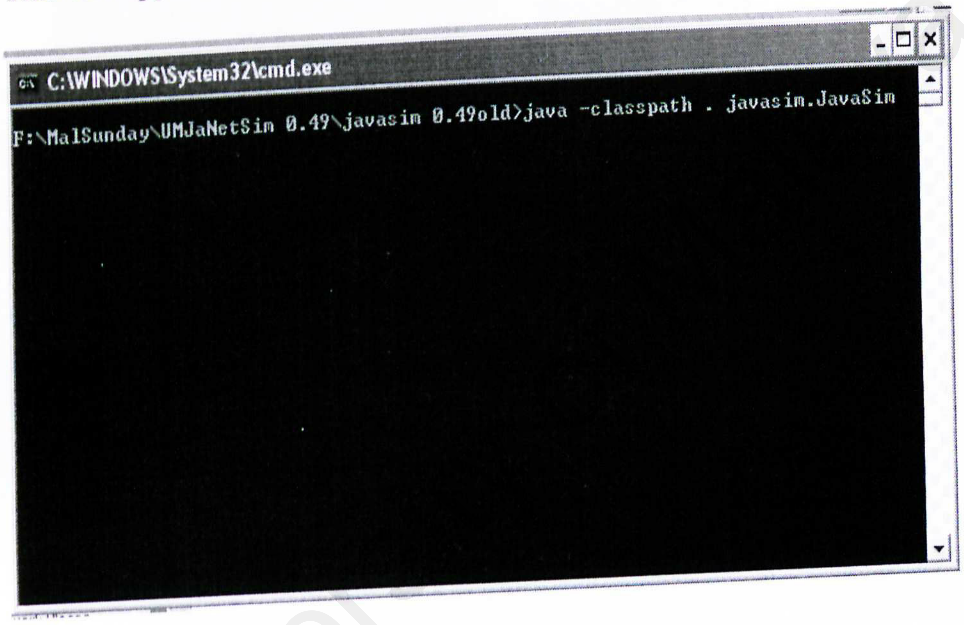
Task	▨	Milestone	◆	External Tasks	▨
Split	....	Summary	▬	External Milestone	◆
Progress	▬	Project Summary	▬	Deadline	⇩

USER MANUAL

1. Java has to be installed in order to run the simulator.
2. To run the simulator, click the 'go' batch file (Figure A1) which is located in 'javasim 0.49old' folder.

Figure A1 : 'go' Batch File

3. This will appear on the screen :



```
C:\WINDOWS\System32\cmd.exe
F:\MalSunday\UMJaNetSim 0.49\javasim 0.49old>java -classpath . javasim.JavaSim
```

Figure A2 : cmd.exe

4. The simulator will then appear on the screen (Figure A3).

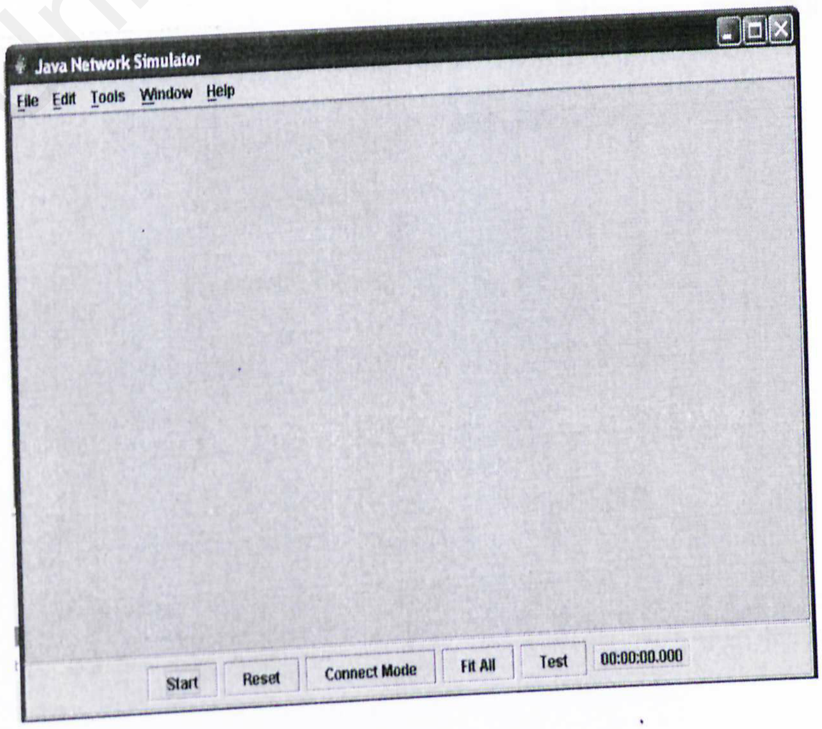


Figure A3 : UMJaNetSim Network Simulator

<u>Setting up a simple topology with self-similar component as traffic source</u>

1. To set up a simple topology such as the one shown in Figure A4, the components should first be created. The topology in Figure A4 consists of the following components :
   - (i) Two VBR Self-similar application components (vbr1 and vbr2)
   - (ii) Two Generic BTEs (bte1 and bte2)
   - (iii) Two Generic Links (l1 and l2)
   - (iv) One ATM Generic switch (atm)

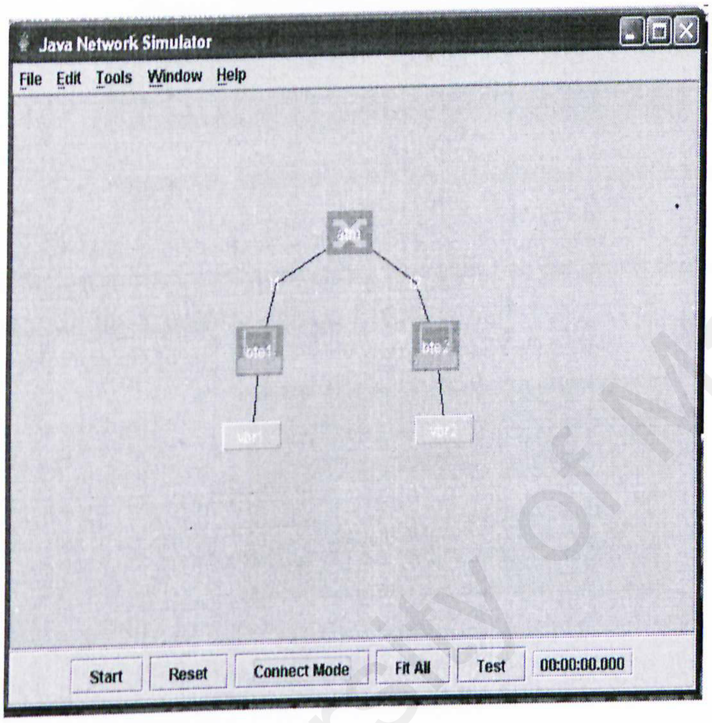


Figure A4: Simple topology

2. To create a component, right-click the mouse. The following pop-up will appear on the screen:

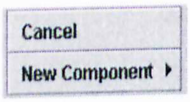


Figure A5: Pop-up

3. To create a VBR Self-similar component, select the "VBR Self-Similar Application" from the "Application" list which is under "New Component" (Figure A6).

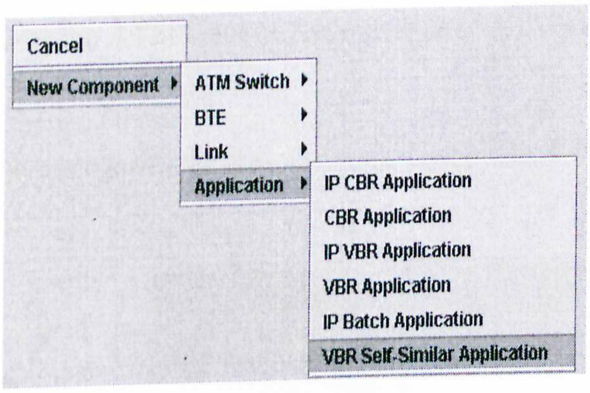


Figure A6: Selecting the VBR Self-Similar Application

4. The following pop-up requesting the name to be given to the component will appear on the screen as shown below :

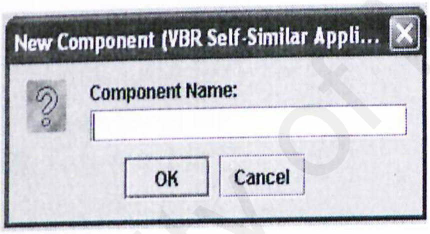


Figure A7: Name request

5. Once the name has been entered and the "OK" button is pressed, the VBR Self-similar component will appear on the screen :

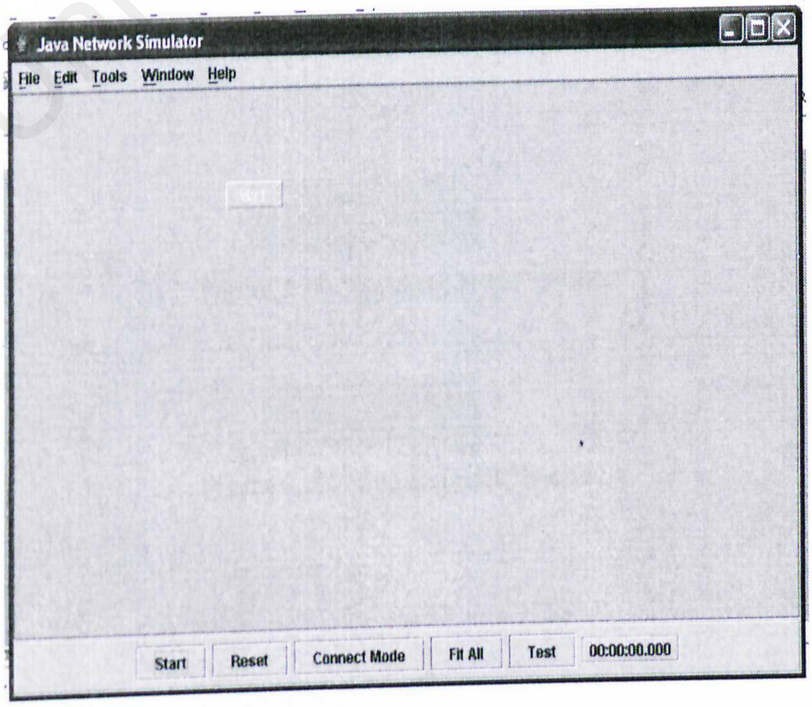
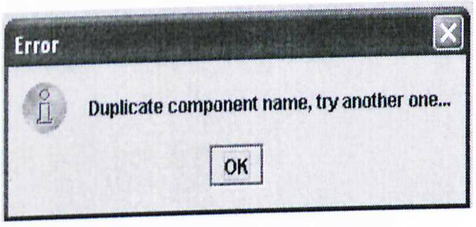


Figure A8: Component created

6. Similarly, create the rest of the components from "New Component" as mentioned in points 1(i) to 1(iv)
   (i) For the two BTE components, select "BTE Generic" from "BTE"
   (ii) For the two Link components, select "Generic Link" from "Link"
   (iii) For the ATM Generic Switch, select "ATM Generic" from "ATM Switch"

7. No two components should have the same name. The following error message will appear :



Error

Duplicate component name, try another one...

Figure A9: Error message

8. Once all the components have been created, they should be connected :
   (i) vbr1 should be connected to bte1,
   (ii) bte1 should be connected to l1,
   (iii) l1 should be connected to atm,
   (iv) atm should be connected to l2,
   (v) l2 should be connected to bte2, and
   (vi) bte2 should be connected to vbr2

9. Connecting is done by first clicking the "Connect Mode" button (Figure A10), and then clicking the mouse on the middle of a component and 'dragging' the mouse to the next component it is to be linked to and clicking the mouse again. A 'white line' will appear between the two components, and the components will then be 'connected'. Once all the components have been connected, click the "End Connect" button (Figure A11).

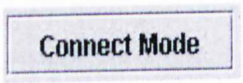


Figure A10: "Connect Mode" button

Figure A11: "End Connect" button

10. The properties of the components should then be set. To send traffic from vbr1 to bte2, right click the vbr1 component and select "Properties" from the list that appears (Figure A12).

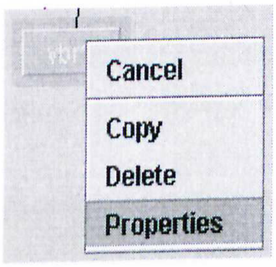
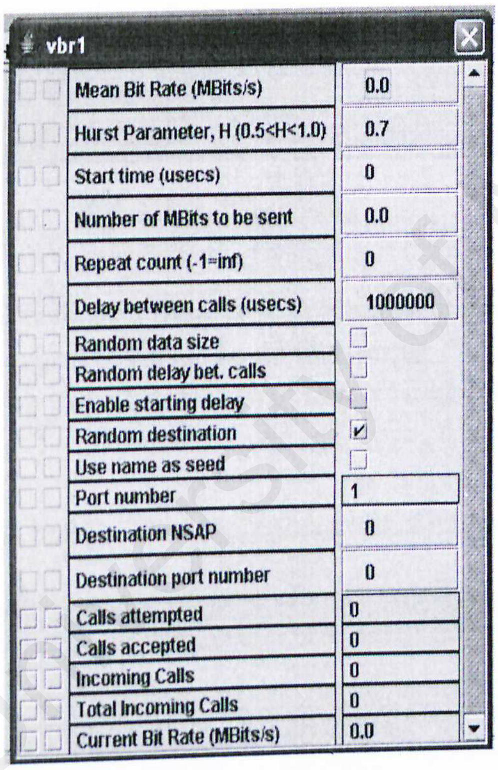


Figure A12: Selecting "Properties"

11. The property box of vbr1 will then appear :


vbr1

Mean Bit Rate (MBits/s)	0.0
Hurst Parameter, H (0.5<H<1.0)	0.7
Start time (usecs)	0
Number of MBits to be sent	0.0
Repeat count (-1=inf)	0
Delay between calls (usecs)	1000000
Random data size	☐
Random delay bet. calls	☐
Enable starting delay	☐
Random destination	☑
Use name as seed	☐
Port number	1
Destination NSAP	0
Destination port number	0
Calls attempted	0
Calls accepted	0
Incoming Calls	0
Total Incoming Calls	0
Current Bit Rate (MBits/s)	0.0

Figure A13: Properties of vbr1

12. Fill up the following properties:
    (i) Mean Bit Rate – choose a positive value
    (ii) Hurst parameter – choose any value between 0.5 and 1.0
    (iii) Start time – any positive integer, e.g. 1
    (iv) Number of MBits to be sent – any positive value
    (v) Repeat count – set it at 1
    (vi) Delay between calls – any positive integer
    (vii) Destination NSAP – set it at 1

13. For the ATM Generic switch, click on the "Manage" button which is next to the "Route Table" property (Figure A14) on the property list. The Route Table as shown in Figure A15 will appear.

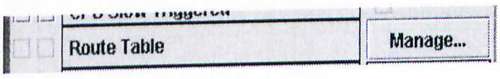
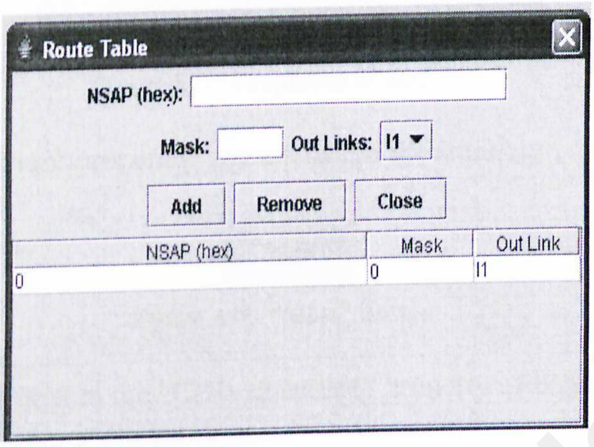


Figure A14: "Manage" button

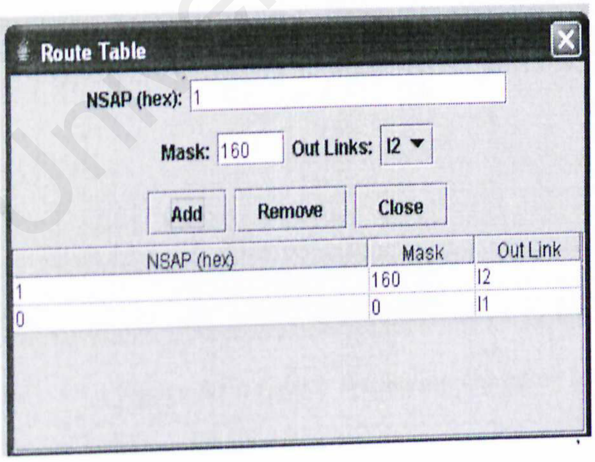


NSAP (hex)	Mask	Out Link
0	0	l1

Figure A15: Route Table

14. In order to send cells/data from vbr1 to bte2, there has to be a path from the ATM switch to bte2, which is via l2. To do this, under "NSAP(hex) : " enter '1'. Under "Mask" enter any value between (including) 1 and 160. Under "Out Links : " select l2.

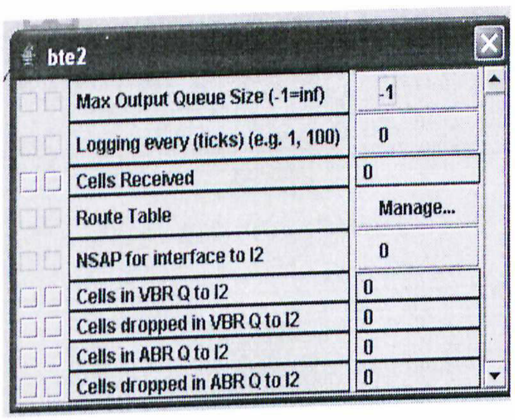
15. Click the "Add" button. The screen should then be updated :



NSAP (hex)	Mask	Out Link
1	160	l2
0	0	l1

Figure A16: Route Table updated

16. Next, on the property list of bte2 (Figure A17), set the "Logging every" property as 1. In order to send data from vbr1 to bte2, both components should have the same value for the NSAP. The "NSAP for interface to l2" should therefore be set to 1.



bte2

Max Output Queue Size (-1=inf)	-1
Logging every (ticks) (e.g. 1, 100)	0
Cells Received	0
Route Table	Manage...
NSAP for interface to I2	0
Cells in VBR Q to I2	0
Cells dropped in VBR Q to I2	0
Cells in ABR Q to I2	0
Cells dropped in ABR Q to I2	0

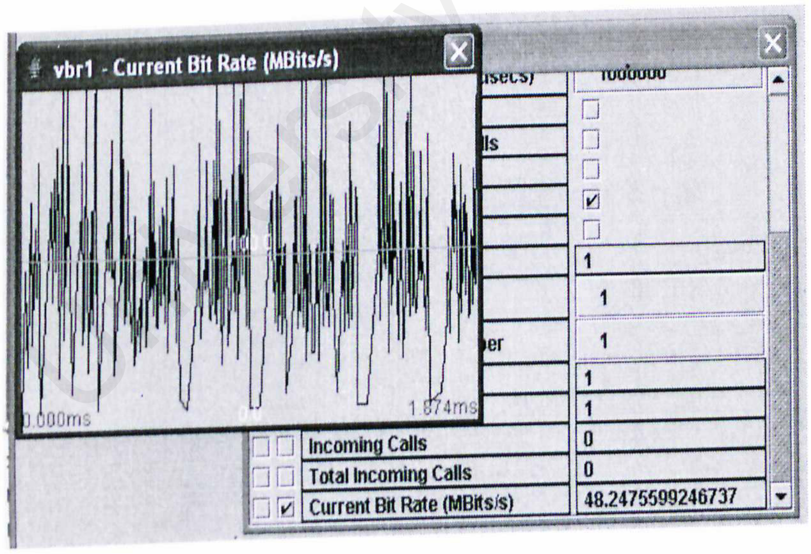
Figure A17: Properties of bte2

17. Once the relevant properties are set, to run the simulator, click the "Start" button.

Figure A18: "Start" button

18. The numbers next to the "Cells Received" property of both the ATM switch and BTE2 should increase with time.

19. The values of the "Current Bit Rate" property of VBR1 should change constantly. To view the changing bit rate in the form of a graph, click the check box next to the "Current Bit Rate" property. A graph should appear (Figure A19).



Incoming Calls	0
Total Incoming Calls	0
Current Bit Rate (MBits/s)	48.2475599246737

Figure A19: Graph displaying changing bit rate

20. To pause the simulation at any time, click the "Pause" button. To reset the values at any time, click the "Reset" button.

Figure A20: "Pause" button

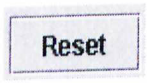


Figure A21: "Reset" button