MEE08:36

Network Emulation, Pattern Based Traffic Shaping and KauNET Evaluation

Abdul Azim

Zafar Iqbal Awan

This thesis is presented as part of Degree of

Master of Science in Electrical Engineering

Blekinge Institute of Technology
November 2008

Blekinge Institute of Technology
School of Engineering
Department of Telecommunication
Supervisor: Dr. Markus Fiedler

ABSTRACT

Quality of Service is major factor for a successful business in modern and future network services. A minimum level of services is assured indulging quality of Experience for modern real time communication introducing user satisfaction with perceived service quality. Traffic engineering can be applied to provide better services to maintain or enhance user satisfaction through reactive and preventive traffic control mechanisms. Preventive traffic control can be more effective to manage the network resources through admission control, scheduling, policing and traffic shaping mechanisms maintaining a minimum level before it get worse and affect user perception. Accuracy, dynamicity, uniformity and reproducibility are objectives of vast research in network traffic.

Real time tests, simulation and network emulation are applied to test uniformity, accuracy, reproducibility and dynamicity. Network Emulation is performed over experimental network to test real time application, protocol and traffic parameters. DummyNet is a network emulator and traffic shaper which allows nondeterministic placement of packet losses, delays and bandwidth changes. KauNet shaper is a network emulator which creates traffic patterns and applies these patterns for exact deterministic placement of bit-errors, packet losses, delay changes and bandwidth changes. An evaluation of KauNet with different patterns for packet losses, delay changes and bandwidth changes on emulated environment is part of this work.

The main motivation for this work is to check the possibility to delay and drop the packets of a transfer/session in the same way as it has happened before (during the observation period). This goal is achieved to some extent using KauNet but some issues with pattern repetitions are still needed to be solved to get better results. The idea of history and trace-based traffic shaping using KauNet is given to make this possibility a reality.

ACKNOWLEDGEMENT

First and foremost we would like to thank our supervisor Dr. Markus Fiedler, who has shown large and consistent interest in our project during the times. Our numerous scientific discussions and his many constructive comments have greatly improved this work.

We would like to thank Dr. Johan Garcia and Per Hurtig from the Karlstad University who provided us practical guidance for our thesis work. We are also thankful to Dr. Patrik Arlos, Fredrik Erlandson and Tahir Nawaz Minhas for their support.

We would like to show our gratitude and love for our parents who have always motivated and appreciated our efforts. Finally we are really thankful to Almighty Allah without whose blessings none of this has been possible.

List of Figures

Chapter 1

INTRODUCTION

1.1 Motivation

Conventional use of Internet was based on availability and use of Internet resources for conventional communication like email, messages and transfer of textual data. Delivery of data was not as time critical as in modern real time communications. More crucial real time communications include transfer of audio and video data over Internet such as IP-Telephony and live video conferences plus live video streaming. There are many other services where response time over Internet has a huge effect on user satisfactions like e-commerce industries Global Information System (GIS) and downloading of data. There is a tremendous increase of Internet users and use across the globe which has increased the amount of Internet traffic. There are different levels of users acquiring Internet service depending upon the Service Level Agreement (SLA) between service provider and end user. In order to meet these different levels we need to control the traffic over Internet which is provided through different QoS mechanisms.

Traffic shaping is used to control network traffic to provide guaranteed performance, lower latency and increase usable bandwidth by applying delay on packets. Additional delay is imposed on packets to such that they conform to some predetermined constraint. Basic function of a traffic shaper is to provide delays on packets to ensure guaranteed service for different traffic classes. Network traffic shaping uses packet classification, handling of queues, scheduling of packets for transmission. Queues are used to buffer traffic packets and packets are transmitted with fixed delay time depending on the traffic classes to ensure better services utilization.

In a trace based traffic shaping, previous behaviour of network traffic is analysed. Depending upon this behaviour, the strategy for future traffic shaping is prepared. However, to reach this aim the real network behaviour needs to be imitated as realistically as possible. This idea is also known from trace based simulation. In trace-based simulation some predefined or pre-calculated values of traffic are used for simulation and results are collected for future use.

KauNet emulation system provides pattern-based emulation. In KauNet patterns can be created from collected traces, previous simulations or analytical expressions. Patterns can be used to match with packets or streams and perform shaping on them as required.

1.2 Objective

1. To review literature about QoS mechanisms.
2. To review the literature on Linux Traffic Control (TC) and traffic shaping.
3. To review the different approaches used for traffic shaping and provide overview of them.
4. To review the literature about KauNet.
5. Evaluation of KauNet for packet losses, delay and bandwidth limitation patterns.
6. Propose the solution for history and trace-based traffic shaping.

1.3 Organization of the thesis

This thesis report is organized as, Chapter 2 gives a background of thesis work, a detail description of QoS and Traffic Engineering. Chapter 3 describes the Linux Traffic Control and Traffic Shaping along with literature review of different shapers and emulators. Chapter 4 includes the description of KauNet design, DummyNet, IPFW and Pattern Generator. In Chapter 5 experimental environment of the thesis work is described. Chapter 6 includes experimentation for evaluation of KauNet, results and analysis of the results. In Chapter 7 the solution for the history and trace-based traffic shaping using KauNet is given. Chapter 8 contains thesis conclusion and future work.

Chapter 2

BACKGROUND

2.1 Quality of Service

Quality of Service (QoS) is to provide different traffic classes over Internet to different users depending upon the services and the willingness to pay. Present Internet without QoS tries to provide best effort services but there is no guarantee of timely delivery and packet loss during transmission. Simultaneously QoS guarantees a bound on delay and loss rate and minimum bit rate within a service class. There could be many services classes depending upon which guarantees are to be provided. There are different approaches providing QoS over Internet. One approach is over provisioning, to make the bandwidth abundant and there will not be any need to provide QoS without taking extra measures. Implementation of network resources to provide bandwidth at very large scale doesn't seem to exist soon, thus we have to adopt some mechanisms to provide QoS [1]. Mechanisms to provide QoS are Type of Service (TOS) Integrated Services Model (IntServ), Differentiated Services Model (DiffServ), Multiprotocol Label Switching (MPLS), Traffic Engineering and Constrained Based Routes [2][3].

2.1.1 Type of Service

The first idea of providing QoS was to use the field of IP precedence and Type of Service from IP packet header. There are two parts of the packet field named as ToS, three bit precedence and three bit ToS field. The value from 0 to 7 was placed in the precedence field indicating the priority of the packet from lower value to best. Three bits used for Type of Service were to specify handling of packets with more critical traffic parameters like delay, throughput and reliability. This approach was not very successful to provide Quality of Service due to the lacking of architectural framework to use ToS filed [3].

2.1.2 Internet Stream protocol

Internet stream protocol was made to support real time traffic communication. It has never been introduced for public use. It has been used for research purpose only and experiments showed that it was very useful for voice data transfer. It was also known as IP V5 and it failed because it didn't support the administrative hierarchy of Internet [3] [4].

2.1.3 Integrated Services Model

In Integrated Service Model, Resource Reservation Protocol (RSVP) is used for resource reservation before transfer of data. Applications find the path first and then reserve resources at every hop within the path to provide QoS. In Integrated Services Model reservations are made on a per-flow basis and every hop within the path should be RSVP protocol enabled. There are four components of Integrated Service Model: RSVP, Admission Control, Packet

Classifier and Packet Scheduler. RSVP is used as signaling protocol to reserve the resources, admission control is used to decide which packet should be accepted or dropped. The Packet Classifier identifies the flows that different packets belong to sort packets per flow and Packet Scheduler decides the order of transmission of packets [2].

Integrated services model was not very famous to provide Quality of Service in bigger networks. To reserve resources per flow basis in bigger networks can be a big problem because of large number of traffic flows. It is not scalable for every node to be RSVP enabled and it is not administratively feasible to provide end to end Quality of Service using Integrated Services Model [2] [3].

2.1.4 Differentiated Services model

Internet architecture provides an end to end framework and in Quality of Service we have to provide end to end Quality of Service to users. In differentiated service model architectural framework of Internet is used to provide QoS. Internet is made by the aggregation of different autonomous systems. As Internet is combinations of many different entities which are called local clouds of network, these are not homogenous from administrative as well as resources point of views. The idea of reserving end to end resources for providing QoS over edge to edge network becomes inefficient.

In differentiated services model resources are reserved locally for every cloud and interact with its neighboring networks. Quality of Service building blocks are formed in DiffServ model and combine together to provide end to end Quality of Service through edge to edge resource reservations. Traffic aggregates are made within a cloud depending upon the traffic type, application type, port number or customer service level instead of flows. There are local rules for every cloud which are not visible to outer world. Networks are homogeneous in nature within a cloud from the perspective of technology, administration and availability of bandwidth. The neighboring clouds can have totally different technology, administration and bandwidth limits. Quality of Service implementation is applied at edges of every cloud through marking, policing and shaping depending upon the priority of traffic aggregates. Per-hop behavior is defined in DSCP (differentiated service code point) using the traffic class octet of IPv6 or the ToS filed of IPv4 packet format.

2.2 Traffic Engineering

The concept of traffic engineering has been introduced to provide network resource optimization. Traffic Engineering is a general frame work having more traffic oriented requirements with economical and reliable network resource utilization [5]. We can provide better utilization of network resources through network traffic engineering along with DiffServ. Traffic engineering has been applied with MPLS to provide Quality of Service.

Traffic engineering controls network traffic to get maximum utilization of network resources to assist better Quality of Service. Traffic control is divided into preventive traffic control and reactive traffic control [6].

2.2.1 Reactive traffic control

Reactive traffic control is used to handle the traffic depending upon the current network situations rather than using preventive actions. Reactive traffic control deals more with loss sensitive services along with optimized throughput. Traffic control mechanism such as Feedback Flow Control is used to regulate traffic flow [10].

2.2.2 Preventive traffic control

Preventive traffic control tries to avoid situations of congestion. Network resources are managed in a way that unacceptable conditions should not be created and better utilization of network resources [7] was achieved. Preventive traffic control can be helpful for less variable bit rate transmission but it can be less useful for variable bit rate transmission of bursty data traffic [7]. Preventive traffic control can be provided through effective admission control, scheduling, policing and shaping.

2.2.2a Admission control

Traffic is controlled at the entry point of the network by accepting or denying the connection request during connection setup time. Acceptance of an incoming connection request depends on the actual bandwidth utilization of ongoing (communicating) connections. If this utilization is less than the threshold then connection will be accepted; otherwise it will be rejected. Admission control is usually used for real time communication traffic connections [7] [8]**.**

2.2.2b Scheduling

Network traffic schedulers are used to schedule the network traffic to provide Quality of Service by handling the packet queues. Scheduling can be done by implementing different scheduling algorithms along with use of dropping algorithms [9]. Network traffic packets can be scheduled on the basis of different required parameters like delay, cost, loss rate etc.

2.2.2c Policing

Traffic policing is used to limit the bandwidth utilization for more critical network traffic to provide QoS. In traffic policing excessive traffic is dropped when traffic reaches to the maximum level. Policing doesn’t require handling of queue or scheduling function for retransmission. Policing is usually applied where packet loss can be tolerated. Policing avoids delays because it doesn’t have to handle traffic queues for QoS.

2.2.2d Shaping

Traffic shaping is used to control network traffic to provide QoS by applying low latency and uniform jitter. The other usage of shaping is to mimci/emulate network behaviour in order to study its impact e.g. on the behaviour of protocols, applications etc. It deals with packet classification, handling of Queues and scheduling of packets for transmission. Shaping is commonly used mechanism to provide QoS to fulfill service level agreements.

Chapter 3

TRAFFIC CONTROL AND TRAFFIC SHAPING

3.1 Linux Traffic Control (TC)

Linux is a very strong operating system that is being used by network and internetworking communities as well as by end users. Linux provides a group of tools for traffic engineering over a network to obtain required behavior of network. Traffic Control (TC) [11] is one of the strongest tools to manage traffic packets. Quality of Service can be provided through Linux traffic control by defining the behavior of the kernel on packet transmission. TC uses queuing disciplines to handle packets to provide different treatment by the kernel during transmission i.e. packet delays, priorities and packet dropping etc. There are two types of Queuing disciplines in Linux traffic control [11].

1. Classless Queuing Disciplines
2. Classfull Queuing Disciplines

3.1.1 Classless Queuing Disciplines

These queuing disciplines don’t have their sublevels. They can contain the data for entire interface to apply delay or rescheduling of packets for transmission to provide QoS.

Fifo_fast

This queuing discipline handles packets on the basis of packet order to enter the queue. First arrived packet will leave the queue first. No special care is given to any packet within the queue. FIFO-fast has three different bands named as band 0, 1 and 2. Kernel decides with the help of ToS field to place packet in its appropriate band. Every band has first in first out treatment to the entering packets. Band 0 has the highest priority i.e. packets within a band 0 will be scheduled to transmit before1 and 2.

Token Bucket Filter

Token bucket filter is a simple queue disc that permits some short bursts that exceed the defined rate limit. It allows the packets to pass which are within the maximum defined rate. Token bucket filter acts like a shaper to slow down the traffic rate over an interface. TBF consist of a bucket with a specified bucket size. Bucket is continuously filled with tokens having a specific token rate. Tokens are deleted from the bucket depending upon the arriving rate of data packets.

- If the rate of incoming tokens and incoming data are equal, there will not be any delays to data packet within a queue.

- If arrival rate of tokens is greater than the packet rate, there will be some extra tokens present in the bucket. These extra tokens will be used for short bursts to pass through the queue which exceed the defined packet rate.
- If the rate of arriving packets is greater than the token rate, the shaper will run out of tokens and data packets will start dropping. This particular last case will allow us to control the sustainable traffic data rate within the defined limit.

Parameters

Limit or Latency: Contains maximum number of B in a queue waiting for token or latency parameter can also be set i.e. maximum amount of time that a packet can be in a queue.

Burst: It contains maximum amount of B that can get a token without delay.

MPU: It is minimum size of packet to get a token.

PeakRate: Specifies how fast token can be deleted from bucket.

Configuration

```
# tc qdisc add dev eth0 root tbf rate 300 kbps latency 100 ms burst 1540
```

The line above slows down sending to a rate that lead to a queue in Linux, where we can control it to a limited size.

Stochastic Fairness Queuing

It's simpler and less accurate queuing discipline as compared to other fair queuing algorithms, it needs less calculations to implement. SFQ is used to divide the traffic into different FIFO queues and these queues are assigned to different TCP traffic flows and UDP traffic streams. SFQ uses hash function to separate the traffic into separate FIFO's which are dequeued in a round robin fashin. In fact it doesn't assign a queue to each flow. If there is large amount of traffic available to transmit from an interface then SFQ can be useful otherwise SFQ will have no effect.

Parameters

Perturb: Number of seconds during which hashing has to reconfigure itself.

Quantum: Amount of bytes that a stream can dequeue before the next queue. It must be set to the size more that the MTU.

Limit: It contains maximum number of packets in a queue. After this SFQ starts dropping packets.

Configuration

```
# tc qdisc add dev eth0 root sfq perturb 10
```

Hashes are needed to be reconfigured, if the value is not set then hash will never be reconfigured. The value 10 in our example recofigure hash after every 10 seconds. It is considered to be a good value for reconfiguration.

3.1.2 Classfull Queuing Disciplines

Classfull queuing disciplines have a hierarchical structure, a queuing discipline can have sub levels (classes) and these subclasses can have further subclasses. The concept of parent and child has been used in classfull queuing disciplines**.** A class can have a queue disc as a parent or even a class as a parent, a class having no child is known as leaf class. By default a FIFO disc is attached to every class but other queuing disciplines can be added to all classes and to their subclasses in place of FIFO. Classifiers and filters are used to guide the packets to their appropriate class where these can get the special treatment depending upon the queuing discipline used by that class.

Classfull queuing disciplines can be useful to handle different types of traffic providing different treatment through scheduling, policing and traffic shaping. There is an egress root queue disc attached with every interface. There can be an ingress queue disc applying policing on incoming traffic.

The PRIO Queue disc

The PRIO disc divides traffic into different classes, classes are based on the commands to configure filters, by default there are three classes containing FIFO qdisc in every class. PRIO qdisc doesn’t really shape the traffic because it just adds FIFO classes to it. More appropriate queue disc can be added on these classes to apply shaping. PRIO qdisc is a work conserving queuing discipline, no delay will be applied if resources are available to transmit data. Class 1 will always have the highest priority; class 2 and 3 will only be used if there is no packet to send in class 1.

Parameters

Bands: The number of classes to create, there are three classes by default.

Priomap: If we don’t configure TC filters to classify traffic, the PRIO disc uses TC_PRIO to enqueue traffic.

Configuration

```
              1:
           /  |  \

        /     |     \

     1:1     1:2    1:3
      |       |      |

      |       |      |

     50:     60:     70:

     SFQ     SFQ     SFQ
```

```
# tc qdisc add dev eth0 root handle 1: prio

# tc qdisc add dev eth0 parent 1:1 handle 50: sfq

# tc qdisc add dev eth0 parent 1:2 handle 60: sfq

# tc qdisc add dev eth0 parent 1:3 handle 70: sfq
```

Prio qdisc is added using the first command having root as parent with handle ‘1’, bydefault three classes are created with handles 1:1, 1:2 and 1:3. The next command is used to add SFQ qdisc with parent 1:1 and handle 50. The next two commands add SFQ with handle 60 and 70 respectively.

Class Based Queueing Shaping [12]

Class Based Queueing (CBQ) is a famous, most complex and difficult to understand queuing discipline. CBQ Shaping is done using idle time calculations which depends on dequeue time and bandwidth of the link. It is very difficult to find the idle time of a link as there are many factors that can affect the calculation of exact idle time i.e. time during hardware layer requests, bad driver of device or device on slightly different protocols like PPP device over Ethernet. Some times CBQ acts wrongly with no obvious reason.

CBQ drives idle time by calculating time between average packets. Effective idle time is measured through exponential weighted moving average (EWMA); it gives exponentially more weight to most recent packet than previous packets. Then it subtracts the calculated idle time to find ‘avgidle’. If link is empty then it can have very large avgidle and it decrease with the increase in amount of traffic and an overloaded link can have negative idle time.

CBQ acts like a PRIO queue to priorities the traffic in to different classes and WRR is applied to all traffic classes starting from highest priority classes to lower priority classes. Classes having same priority level are grouped together, these classes within the priority are tried to dequeue one after another.

Configuration

```
              1:          root qdisc
              |
             1:1           child class
             /   \
            /     \
          1:3    1:4       leaf classes
           |      |
          30:    40:       qdiscs
         (sfq)  (sfq)
```

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

Hierarchical Token Bucket [12][13]

Hierarchical Token Bucket denotes a hierarchical queuing discipline with token bucket filter implemented as a classfull hierarchal queuing discipline. It works like CBQ but it doesn’t use idle time calculation to shape traffic. If some bandwidth is allotted to CBQ, it will not use more bandwidth even if it is available. In HTB, some amount of bandwidth can be specified for a class and even it can be specified how much bandwidth a class can use more than its defined bandwidth if it is available. Default class can be specified that can take the rest of bandwidth which is not reserved for any other class. Parameters of HTB are almost same as of CBQ except some additional parameters to specify the extra bandwidth that a class can use if needed.

Configuration

```
# tc qdisc add dev eth0 root handle 1: htb default 30
# tc class add dev eth0 parent 1: classid 1:1 htb rate 6mbit burst 15k
```

The command is used to add an HTB with parent as root and handle ‘1:’. The unclassified traffic is routed to 30:, which has little bandwidth but can borrow erverything that is left over. The second command is used to add 1:1 as a handle with parent 1: with configured rate 6mbit and burst size of 15k.

3.2 Network Traffic Shaping and Emulation

3.2.1 Traffic Shaping

Traffic shaping is used to control network traffic to provide Quality of Service, low latency, and increase usable bandwidth by delaying packets. It deals with packet classification, handling of Queues, scheduling of packets for transmission. The basic function of a traffic shaper is to insert delays on packets to ensure guaranteed service for different traffic classes. Queues are used to buffer traffic packets which are transmitted with fixed delay time depending on the traffic classes to ensure better services utilization.

3.2.2 Trace-Based Traffic Shaping

In a trace-based traffic shaping, the previous behaviour of network traffic is recorded. Depending upon this behaviour the strategy for future traffic shaping is prepared. Motivations behind trace based traffic shaping are to produce behavior of network traffic exactly as earlier traces. Some traces from real or emulated networks are collected (ideal traces) and analyzed to calculate exact values of traffic parameters e.g. delay, jitter, loss rate and bandwidth utilization. This idea is also known as trace-based simulation/emulation.

3.2.3 Network Emulation

Network emulation refers to an experimental network that can mimic a target network by imposing predefined network effects to traversing traffic. Emulation provides more controlled and reproducible test environment to facilitate protocol testing and performance evaluation. A network emulator can be a workstation or group of connected workstations [16]. Network emulation is famous in research community, as it is difficult to access real networks even if they are accessible, still an emulator can have more controlled and reproducible environment. A network emulator should hold following properties [16]: it has to be open and extensible network environment and capable of testing all type of protocols. It should be scalable in terms of emulation capacity and accuracy. A network emulator should be able to configure differet network topologies and emulation parameters.

3.3 Literature Review of Network Traffic Shaping and Emulation

3.3.1 Dummynet: a simple approach to the evaluation of network protocols [14]

The effectiveness and performance of network protocols can be analysed in operational networks or through simulations. The former approach has its limitations because networks cannot always be configured with desired features, and mostly the actual operating conditions are not easily controllable. Simulations have the advantage of being much more controllable and reproducible. Simulations are however only approximations of the reality. Simulations also have some limitations. It's hard to simulate all the factors that are related to the experiment correctly and perfectly. Nevertheless, the difficulty of setting up a real network with the desired features has stimulated the development of a number of simulators, such as REAL, NetSim and NS.

The experiments on network protocols are performed to determine the behaviour in a complex network made up of many nodes, routers and links, with different queuing polices, queue sizes, bandwidths, propagation delays. Quite often, one of the routers is modified to act as a "flakeway", introducing artificial delays, random packet losses and reordering [14].

The concept of a flakeway proposed a simple and effective approach to insert a model of simplified network in an operational stack, and run experiments on a standalone system. The approach is called Dummynet, which gives advantages of both simulation and real world testing, great control over operational parameters, simplicity, ability to use real traffic generators. Using dummynet one can run experiments quickly and easily by using real world applications like FTP, Telnet, Web browsers as traffic generators. Experiments in dummynet can be done up to the maximum speed supported by the system under use.

The layers in a protocol stack communicate with adjacent layers. The simplest setting usaully consist one or two routers and a pipe. These components can be modeled using a couple of queues rp/pq (router queue/pipe queue). K denotes the maximum size of the rq, B is bandwidth and tp is propagation delay of pipes in case of pq. Traffic exchange between the layers needs the following processing. The packets are inserted in rq according to the maximum size of queue K, depending on the queuing policy used e.g. RED. Packets are then transferred to pq at the rate of B, pq uses FIFO to transfer packet to next layers [14]. Packets remain in pq for tp seconds and then are dequeued to next layers. Random packets losses can also be added at this stage. This simple setting can be used to simulate almost every type of scenario available in literature.

Dummynet can be used in scenarios where reproducibility is unusual. It can be used for debugging purpose. It's very useful in studying the behaviour of new protocols. It can be used for performance evaluation, how the application will work under the given parameters (e.g. bandwidth B, delay T).

Dummynet provides only an approximation of the reality as simulation does. The approximation depends on the granularity and precision of the operating system's timer. In dummynet different events (e.g. extraction of packets from queues/pipes) occur synchronously with respect to the OS's timer. This synchronous behaviour hides or amplifies the real results, as in real systems different events occur asynchronously. The presence of dummynet doesn't change non-deterministic behaviour of a system. Dummynet uses the size of an IP payload, so it’s simple to include link layer overheads if they are known. It’s also possible to simulate link layer compressions, due to which links show variable bandwidths.

3.3.2 NIST Net: A Linux-based Network Emulation Tool [15]

The environment is semi synthetic as it’s a real network implementation with supplementary means of introducing delays and faults. NIST Net is a simple, fast Linux based network emulator. Emulation is a technique for testing networks and it’s a combination of simulations and real live tests. NIST Net can emulate effects like packet loss, duplication or delay, router congestion and bandwidth limitations. On current generation machines NIST Net has been used at line rate with Gigabit Ethernet cards and 622 Mbps on OC-12 interfaces. NIST Net resembles to Sun-OS based Hitbox and its interface is inspired by MOST radio network emulator, otherwise it is completely independent of these two emulators.

NIST Net can act as router which emulates the behaviour of the whole network being a single hop and can apply packet loss and delay jitter to the traffic passing through it. NIST Net works by managing a table of emulator entries i.e sets of specifications. Set of effects that are applied to the packets being matched and third entry is about the statistics that are kept from the previous experiments, used for matching packets passing through the emulator. NIST Net treats traffic on flow basis, thousand of packet entries can be loaded at a time. Then the packets are treated according to the matching rule set. The entries in the table can be controlled manually or programmatically during emulator operation, while replaying recorded delay/loss traces. For matching specifications NIST Net uses values in the fields of higher layer protocols like TCP, UDP, IPIP, ICMP, IGMP etc. Packet matching code works at line rate which makes the NIST Net fast.

NIST Net consists of two parts, a loadable kernel module and a set of user interfaces. The functionality of the kernel is like a loadable module patched and reloaded during runtime without interrupting active connections. It insulates the NIST Net code to a large degree from changes to the base kernel. The user interface allows multiple processes to control the emulator simultaneously. Two user interfaces are provided with the code, a simple Command Line Interface (CLI) and an interactive Graphical User Interface (GUI), which allows controlling and monitoring a large number of emulator entries simultaneously.

NIST Net is famous and has been widely used in academic and research laboratories due to its ease of installation and use. The ability to specify complex network behaviours through some simple and easily understandable statistical parameters have made NIST Net a useful tool. NIST Net has been used for different testing purposes like Voice over IP, mobile network emulation, adaptive video transmission, satellite and undersea radio link emulation, and interactive gaming.

For traffic shaping using NIST Net parameters like mean, standard deviation and linear correlation can be controlled. These parameters can be set to get some desired distributions, packet to packet delay, minimize or maximize the packet reordering. Actual delays imposed depend on the granularity of the clock. Packet loss and duplication are also changeable parameters. For emulating congestion NIST Net prefers Derivative Random Drop DRD over Random Early Discard RED. DRD is a proactive packet dropping strategy that uses the queue utilization as the basis to determine when to drop packets. DRD drops packets with a probability that increases with an instaneous queue length. The probability of packet dropping depends on queue size when a packet arrives. A variety of scenarios can be created using DRD. Because queue length depends on each flow, minimum and maximum thresholds are implemented. Explicit Congestion Notification ECN is also implemented using DRD. Bandwidth limits can also be implemented by imposing limits on queue lengths through DRD parameters.

3.3.3 EMPOWER: Emulating the Performance of wide area networks [16]

EMPOWER is a flexible, scalable network emulator that uses commodity PC and can emulate wide area networks at reasonable cost. It can also be used in wireless research projects providing mobility in wireline nodes. A node in EMPOWER can be configured to work as multiple real network nodes. Using multiple nodes make the EMPOWER scalable for large networks.

EMPOWER consists of a number of workstations in LAN that can be divided in two categories, emulator nodes and test nodes having at least one NIC. All these are interconnected through a fast Ethernet switch. Any type of topology can be built using the same physical topology of the EMPOWER. The number of emulator nodes can be easily increased because the workstations used are commodity PC that run Linux.

The Virtual Device VD is the basic component of the EMPOWER which is used to apply the predefined network conditions on network traffic packets. The VD is a software module that is loaded on each emulator node. A VD is attached to the physical network port in an emulator node, and both share the same IP. The packets are diverted to the VD using routing table in the kernel, to perform specific operations to those packets. Trace driven emulation is also supported by VD, in which a trace file or real network traffic is used as input instead of traffic generated by an application. A VD module is subdivided into six modules, the MTU, delay, bandwidth, loss, bit error rate and out-of-order sub modules. Each sub module is responsible for one aspect of IP traffic. The VD doesn’t affect the packet receiving procedure of an emulator node.

A real network topology can be mapped to a virtual topology with the same number of routers and ports in one or more emulator nodes. Each router in the real network is mapped to a virtual router with same number of ports and IP addresses. Every virtual router is independent of every other router.

The Virtual Router Mapping is a scheme used for emulating network topologies in EMPOWER. The scalability of the system depends on the mapping technique used for mapping original network to the emulator. An efficient algorithm is needed to divide the original network into a number of blocks those can then be mapped to the emulator nodes. An emulator node can have multiple network ports and behave like a host based router. The maximum throughput depends on the number of ports being used for forwarding packets. Throughput decreases as the number of virtual routers increases due to resource competition in emulator node.

Time accuracy is the key to the emulation accuracy of network which affects packet delay, link bandwidth and packet drop rate. EMPOWER is quite accurate in emulating packet latency with single router in an emulator node but when there are more than one router then each router may impose delay to specific packets due to resource competition, which affects packet latency.

3.3.4 Master Shaper [17]

Master Shaper is a shaper that provides web based interface for Internet traffic shaping over Linux. It is developed for the people who know traffic shaping but are not well familiar with Linux traffic control commands and scripting. It makes traffic shaping easier than Dummynet, NIST Net and EMPOWER to implement for a user. Master shaper provides graphical information about traffic statistics like bandwidth utilization. Master Shaper can be used as Linux router or it can act like a bridge. Master Shaper uses HTB, HFSC and CBQ to queue the traffic packets.

HTB (Hierarchical Token Bucket)

HTB can provide the minimum guaranteed bandwidth to a class plus some extra bandwidth that a class can lend from other traffic classes to use if it is available. Behaviour on burst level can also be specified and priorities can be assigned to traffic classes using HTB. If HTB is selected as Queuing discipline in master shaper, the HTB will be used to classify packets and SFQ (Stochastic Fairness Queuing) will be used as leaf class to shape traffic.

Hierarchical Fair Service Curve (HFSC)

HFSC is more suitable for real time communication like Voice over IP (VoIP) and real time applications where services are more delay and jitter sensitive. HFSC specifies the maximum bandwidth that a class can use for communication. HFSC can provide guaranteed minimum bandwidth available for a class and guaranteed maximum delay.

Class Based Queuing (CBQ)

CBQ uses idle time calculations that depend on dequeued time and bandwidth of the link. CBQ drives idle time by calculating time between average packets. CBQ acts like a PRIO queue to prioritise the traffic into different classes and WRR is applied to all traffic classes starting from highest priority classes to lowest priority class. Classes having the same priority level are grouped together, these classes within the priority are tried to be dequeued one after another. Master Shaper uses CBQ if HTB is not available.

Master Shaper can shape outbound as well as inbound traffic. The Intermediate Queuing discipline is used to shape the incoming traffic.

Intermediate Queuing (IMQ) [18]

The Intermediate Queuing device collects all packets and arranges them into queues, packets have to be requeued into a real queue before they can be transmited. IMQ is used for cross device bandwidth sharing and can be very useful for incoming traffic shaping. The Intermediate Queuing discipline is used to collect incoming packets and then apply shaping.

IP table rules can be used to send the incoming and outgoing traffic to queuing device. In this way, the rules for outgoing traffic can be used to shape incoming traffic.

Overview of Master Shaper

Service Level

Parameters of queuing discipline used by Master Shaper are set by service level. In service level it is specified which queuing discipline has to be used and then, its parameters like bandwidth for both incoming and outgoing, priority settings, the queuing discipline used at the leaf class and some other parameters depending upon the queuing discipline being used.

Filters

Filters are used to match the traffic with defined rules. Master Shaper uses IPtables and TC-filter to match the traffic. The status of that filter have to be specified, i.e whether it is active or not while adding a new filter to a Master Shaper. Filter matching can be on the basis of protocols, ports number, protocol flags, sources and destination IP Addresses, TCP flags, packet length, application used, communication layer and time stamp.

Chains

Chains are used to create traffic channels, each chain has a service level defined with it. If there is only one chain then service level can have the maximum bandwidth available. The target definition is used to put the traffic into chains. Master Shaper uses the fall back service level to specify the non matching traffic to follow this. The status of a chain should be specified while defining a new chain along with selecting existent service level, fall back service, target source and destination address which can be any to any.

Pipes

Pipes are used to combine service level, chains and filter all together. Direction of pipes can always be specified as pipes for incoming traffic and pipes for outgoing traffic.While defining a new pipe the status of pipe, direction of pipe, filter, service level and chain can be specified.

Along with above motioned service level, chains, filters and pipes new targets, port numbers, protocols and users can be added. Chains, pipes and bandwidth can be monitored through traffic graphs. Master shaper provides flexible traffic shaping to provide better Quality of Service.

3.3.5 Easy Shape [19]

Easy Shape is a Linux-based bandwidth controller and traffic grapher showing graphs of configured rules and classes. It provides web based interface to set parameters for bandwidth control. Easy Shape uses HTB.ini which is a shell script enhanced from CBQ. HTB [20] is a queuing discipline used in different queuing devices, it's an advanced form of CBQ curing the drawbacks of CBQ. Easy Shape is used to install on Linux router to control the bandwidth for host and network attached to it. Easy Shape can provide these features.

- Web based interface to set bandwidth control rules and view graph.

- Source address, destination address, port number and subnet can be used to apply bandwidth limits.
- Bandwidth “burst” limits can be specified.
- Exception rules can be provided to some specific time for communication.

3.3.6 Conclusion

Network emulation refers to an experimental network that can mimic a target network by imposing predefined network effects to traversing traffic. Emulation provides more controlled and reproducible test environment to facilitate protocol testing and performance evaluation. Dummynet emulator provides an approximation of reality as simulation does. Dummynet doesn’t change non-deterministic behaviour of the system. NIST Net manages a table of emulator entries to provide shaping recorded from earlier traces. Entries can be added or changed manually or programmatically during the emulation operations. In EMPOWER protocols are loaded into kernel without modification and packet specific operations are performed in VD that contains sub modules instead of having tables or patterns. Trace files or real network traffic can be used to perform trace based emulation in EMPOWER. Master Shaper provides web based interface for shaping using Linux Traffic Control. Easy Shape uses HTB.ini that is an advance form of CBQ and it provides time-based rules at certain time set to change bandwidh limits.

Chapter 4

KAUNET DESIGN

KauNet is a network emulator that provides control on bit errors, packet losses, bandwidth, and delay changes. It has developed by Johan Garcia, Per Hurtig and Anna Brunström at Karlstad University, Sweden and it is still in phase of improvement. Using KauNet, reproducible behavior of network along with an exact control on network traffic over Internet can be provided [21].

KauNet is an extension of the Dummynet network emulator, introducing a new facility of pattern generation and management. Patterns can be created from real network traces, simulations and analytical expressions. Patterns are inserted into kernel and can be dynamically changed or switched by the use of command line tools to cope with dynamic events such as during the emulation process. These patterns are matched with traffic packets to provide required behavior in data driven as well as time driven mode. IPFW is used along with Dummynet to make KauNet work for emulation setup and management. Dummynet is capable of controlling bandwidth, apply delays and import packet losses in a probabilistic manner, whereas KauNet can provide exact control over bandwidth changes, delays changes and importing packet losses, additionally it introduces bit errors in deterministic manner. KauNet provides reproducible behavior of network traffic using Pattern Generator, Dummynet and IPFW [21].

KauNet System Design

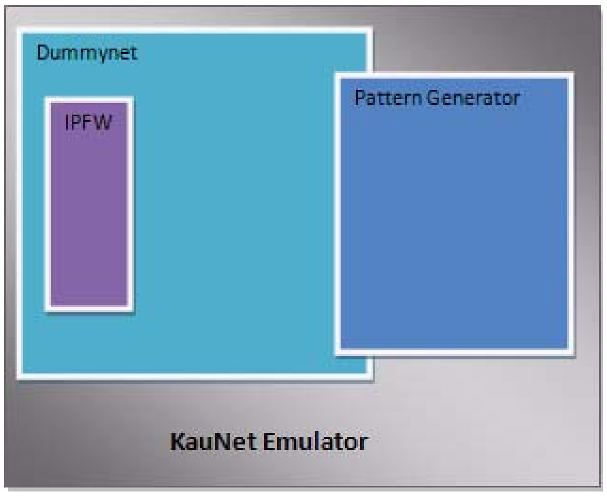

Fig 4.1

4.1 DummyNet

Dummynet is a network emulator, a part of FreeBSD and being used for traffic shaping. Dummynet was originally designed to test networking protocols and later it has been used as bandwidth manager. It is used to simulate queues and bandwidth limitations, delays, packet losses, and multipath effects. Dummynet makes a machine to act as router or bridge and is controlled by IPFW (IP Firewall) commands [22].

4.1.1 Operation

Dummynet is a kernel level operation, packets are fed to Dummynet and IPFW commands are used to make Dummynet operational. IPFW commands will be discussed in the next section.

Packets can be sent to one of the two Dummynet constructs which are:

- Pipe: it simulates a link. Each simulated link can have its own bandwidth, queue size, packet loss rate or propagation delay.
- Queue: it is used to assign a weight to a particular flow, WF2Q+ (variant of Weighted Fair Queuing) is used to share bandwidth for every flow.

Pipes are used to define bandwidth limits and then queues can be used to further define how the bandwidth will be shared among different flows. Each pipe can be configured separately and packets can be forwarded to the appropriate pipe using the IPFW. Different limitations can be applied to different traffic according to the IPFW rules e.g. selection on basis of protocol, addresses, port ranges and interfaces. The traffic which doesn’t pass through pipe or queue simply runs down to the rest of IPFW rules [22].

4.1.2 Kernel Options

The following options in the kernel configuration file related to Dummynet operation have to be set.

IPFIREWALL – it’s required to enable ipfirewall for dummynet

IPFIREWALL VERBOSE – to enable ipfirewall output

IPFIREWALL VERBOSE LIMIT – to limit firewall output

DUMMYNET – to enable dummynet operation

NMBCLUSTER – used to set the amount of network packet buffer, HZ sets the timer granularity, the recommended value of HZ=1000.

In general, FIREWALL and DUMMYNET options are required, additional parameters can be used to increase the number of mbuf clusters (memory buffer, memory management in the kernel) or the sum of bandwidth-delay products and queue sizes of all configured pipes.

4.2 IP Firewall (IPFW)

IPFW is a firewall application sponsored and used by FreeBSD project. It is used for packet filtering and traffic shaping in FreeBSD [21]. IPFW is disabled in the Kernel and it is needed to configure the kernel or dynamically load IPFW to make it work. IPFW is used as a default firewall in FreeBSD [23]. IPFW examines the traffic on packet by packet basis and can handle more than one network interfaces. There are two approaches for firewall, “open” and “closed” approaches. Open means allow everything but block some undesired traffic. However closed mean deny all but allow selected ones. A primary component of IPFW the kernel firewall filter rule processor and accounting facility. The rest of the components are following. Longing facility, “divert” rule for triggering NAT, advance facilities like Dummynet traffic shaping, bridge and IP stealth facility [24].

4.2.1Enabling of IPFW

IPFW can be enabled by two ways, the first is to load the IPFW dynamically and the second one is to custom the kernel and recompile it. The following command can be used as a root to dynamically load the basic IPFW.

```
kldload ipfw
```

Another way to enable IPFW is to add the following line to the kernel configuration file (generic kernal file) and rebuild the kernel.

```
options			IPFIREWALL
```

The default rule of IPFW is to deny all traffic, other rules should be added to allow desired traffic.

4.2.2 Basic syntax of IPFW commands

<command> [<rule #>] <action> <proto> from <source> to

<destination>

All IPFW commands are written with “ipfw” keyword , IPFW “rule number “can be assigned to all rules to add. Every rule has an action associated with it, where action can be a variety of things for example allow or deny. Protocol is defined after specifying the action, then addresses of sources and destinations, “from” and “to” keywords respectively.

4.2.3 Traffic Shaping using IPFW

Traffic shaping can be performed by IPFW along with the use of Dummynet. Probabilistic packet losses, delays and bandwidth limitations can be applied.

IPFW can be used to insert random packet losses with different packet loss probabilities between 0 and 1.

```
add   10 prob 0.7 allow all from any to any  in
```

This will allow 70% and randomly drop 30% of the incoming packets.

For further traffic shaping options, we need to use Dummynet with IPFW. Dummynet can be enabled by adding the following line to the kernel configuration file (generic kernel) and recompile the kernel.

```
Options                 DUMMYNET
```

Now pipes can be configured to filter and control the traffic. A pipe is a traffic shaping rule to handle the traffic in desired manner. Pipes are created using ipfw command along with the pipe keyword. A pipe is created and traffic is redirected to the pipe to apply the configured behavior to control bandwidth, probabilistic packet losses and delays using Dummynet pipes with ipfw [13].

Example:

```
ipfw add allow  all from any to any
ipfw add 10 pipe 200 icmp from 10.0.1.1 to 10.0.2.1 in
ipfw pipe 200 config delay 10 ms
```

The first line allows all the traffic coming from any source to any destination. The second line adds a rule number 10 to create a pipe 200 that allows icmp traffic from 10.0.1.1 to 10.0.2.1. The third line applies 10 ms delay on pipe 200.

4.3 Pattern Generator

The pattern generator tool is developed in C++ to create and manage patterns. The pattern generator is the major component of KauNet. It enhances network emulation provided by Dummynet to introduce deterministic and reproducible behaviour. Patterns can be generated by the command line tool patt_gen, or through patt_gui with graphical interface.

The pattern generator can be used to create different types of patterns defined by different parameters of commands. Patterns can be specified for data driven as well as time drive. In data driven mode the index advances as packet passes, regardless of the time. This mode can be used to specify per packet-behaviour (of KauNet emulation). The time driven mode advances the index as time passes by milliseconds, it can be used to specify the behaviour for specific period of time and to emulate network connectivity outage. Pattern generated by patt_gen are inserted into FreeBSD kernel space for KauNet and then pattern matching is applied on packets when network emulation is started.

Pattern types

As KauNet is an extension to Dummynet, dummynet inserts nondeterministic packet loss, random delay and applies bandwidth restrictions, whereas KauNet introduces control on bit errors along with deterministic packets losses, applies delay changes and bandwidth restrictions with reproducible behavior. Patt_gen can be used to generate patterns for bit errors, packet losses, delay and bandwidth changes and insert them into kernel. Patterns can be specified in Data Driven Mode as well as in Time Driven Mode. In Data Driven Mode index of the patterns move forward per packet arrival whereas in Time Driven Mode index per milisecond basis.

4.3.1 Bit-error patterns

KauNet can insert bit errors into particular positions and change or flip the position of data bits. Bit-error patterns can be generated to control data bits in four different ways. Random bit error patterns insert the bit errors uniformly in all data, Gilbert Elliot bit-error patterns insert bit errors according to the Gilbert Elliot model and fixed position bit-error patterns insert bit errors at fix position in time and data, while fixed interval bit-error patterns insert bit errors to specific interval in data or time.

4.3.2 Packet loss patterns

Packet loss patterns are used to introduce packet losses. Packet loss patterns can be specified both in data-driven and time-driven mode to control packet losses. The atomic unit to advance the position in index is packet instead of bit. Random packet-loss patterns insert the packet losses uniformly in all data. Gilbert Elliot packet loss patterns insert the packet losses according to the Gilbert Elliot model. Fixed position packet loss patterns insert packet losses to fixed position in data and time. Fixed interval packet loss patterns insert packet losses to a specific interval in data or time.

4.3.3 Delay Change patterns

Delay changes patterns are used to control the propagation delay. Delay changes patterns can be specified by command line or by a file. In data-driven mode, the position of the packets can be used to apply delay changes. In time-driven mode the particular position in time (milliseconds) is to be used to apply delay changes.

4.3.4 Bandwidth change patterns

Bandwidth change patterns are used to control the emulated bandwidth change. Bandwidth change patterns can be specified by the command line or by a file as in the case of delay changes. Bandwidth changes for emulated data transfer can be specified in time-driven mode as well as in data-driven mode. In the data-driven mode, the position of the packets can be used to apply bandwidth changes. In the time-driven mode, the particular position in time (milliseconds) is to be used to apply bandwidth changes.

Composite patterns can also be created and assigned to same pipe to control more than one parameters. These patterns are intserted into kernel in compressed form.

4.4 Enhanced pipe syntax in KauNet

KauNet provides pattern generation with exact control of packet losses, bit errors, delay and bandwidth changes being an extension to well known DummyNet emulator. There are some additions in DummyNet pipe configuration syntax along with keywords already defined in DummyNet.

pattern < filename> [timedr|datadr] [nosync]
pipeindex < pipenumber>   ber|pkt|del|bw [timedr|datadr] [nosync]
appendpattern < filename>

pattern < filename>: It contains the name of the pattern to be used.

[timedr|datadr]: it specifies whether the pattern is in time-driven mode or in data-driven mode.

[nosync]: It is used in time-driven mode to specify whether clock should start immediately or whether it should wait till first packet is transferred.The default behavior is to start while the first packet is transferred.

pipeindex   < pipenumber>: This is used to specify that the same pattern file is used by another pipe of specific pattern type.

ber|pkt|del|bw: This is used to specify the type of pattern, i.e. for bit error, packet loss, delay change or bandwidth change.

appendpattern < filename>: It is used to specify if there is any other pattern to append after a pattern is finished, the default behavior is to repeat pattern from its starting position.

Chapter 5

EXPERIMENTAL ENVIRONMENT

The experimental network design consists of three machine named as Host A, Host B and KauNet Emulator. KauNet Emulator is used as a gateway between Host A and Host B. KauNet software is installed on KauNet emulator machine. Experiments have done using ICMP packets generated by ping. Host A sends packets to Host B passing through KauNet emulator to match with generated patterns.

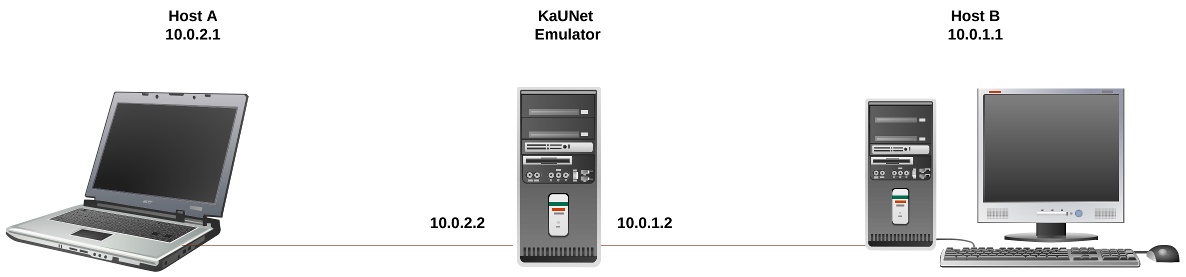

Fig 5.1: Experimental Environment

5.1 Requirements for KauNet

a) FreeBSD 6.0
b) IPFW
c) Dummy Net
d) KauNet machine should be configured as a gateway

5.2 Machines used in Experiments

In the following the specification of the systems used in the experiments is give.

Host A
Intel Pentium Mobile 1.3 GHz processor
512 MB RAM
Intel Pro/100 Ethernet Card
FreeBSD 6.0

Host B
Intel Core ™2, 2.4 GHz processor
3 GB RAM
RealTek 10/100 base TX Ethernet Card
FreeBSD 6.0

KauNet emulator machine
AMD Athlon™ Dual core 2.6 GHz
2 GB RAM
Em1: Intel Pro/1000 Network Connection
Em2: Intel Pro/1000 Network Connection
FreeBSD 6.0

5.3 Installation and Configuration Manual for KauNet

1) Make sure that the kernel source tree is installed in /usr/src/sys and the sbin source tree is installed in /usr/src/sbin

2) Extract the KauNet files, provided in kaunet.tar.gz, in /usr

NB! This will overwrite the following configuration files:

a) /usr/src/sys/conf/files

b) /usr/src/sys/conf/options

Make sure to backup these if you have made modifications to them.

3) Enter the kernel configuration directory

```
cd /usr/src/sys/i386/conf
```

4) Make a copy of the default kernel configuration file

```
cp GENERIC MYKERNEL
```

5) Modify "MYKERNEL" to include the following lines

```
options         IPFIREWALL

options         DUMMYNET

options         KAUNET
```

6) Configure the kernel according to the configuration file

```
/usr/sbin/config MYKERNEL
```

7) Compile and install the new kernel

```
cd  ../compile/MYKERNEL

make cleandepend; make depend;  make; make install
```

8) Restart computer

9) Compile and install the new ipfw:

```
cd  /usr/src/sbin/ipfw

make; make install
```

10) Copy the patt_gen on to the KauNet emulator machine
11) Compile patt_gen by gcc -o patt_gen patt_gen.c
12) execute it with . /patt_gen

KauNet is ready to use, now you can generate pattern and perform your required experiments.

5.4 Problems and difficulties faced during KauNet install and configuration

a) It took much time to get familiar with FreeBSD operating system environment.

b) Problems were faced during custom and building KauNet kernel, showing the Error: “Error code 1: Stop in usr/src/sys/amd64/compile/KAUNETKERNEL.

c) You must build a kernel first”, while executing the “make” command.

d) While make build kernel (KAUNETKERNEL), the error, “make: don’t know how to make buildkernel.stop” was shown.

We faced these problems while installing the KauNet and tried many solution spending a lot of time on www.google.com. Later, we come to know that there were some problems with the source code of the pattern generator. We got the latest version of the pattern generator and follow the instructions to install.

Chapter 6

EXPERIMENTS AND RESULTS ANALYSIS

6.1 Packet Loss Data-Driven Mode

In data-driven mode patterns are created for Fixed Position Packet Loss, Fixed Interval Packet Loss, Random Uniform Packet Loss and Gilbert-Elliot Packet Loss. Patterns have been applied to a given number of packets as specified in the pattern typically mutliple of 40 packets, e.g. 80 ICMP packets generated by ping. The experiment is repeated 40 times to check effect of the pattern. The ping response was recorded and the packet sequence number showed the presence or absence of packet. In the graphs '0' shows the packet is dropped whereas '1' represents presence of the packet.

6.1.1 Fixed Position Packet Loss

Fixed position patterns allow specified individual packets to be dropped. Following pattern is created to insert packet losses at the positions 5, 10, 15, 20, 25, 30 and 35.

```
./patt_gen -pkt -pos test1.plp data 40 5,10,15,20,25,30,35

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config pattern test1.plp
```

i) Number of packets equal to pattern size (40/40)

Fixed Position Packet Loss - runs 1, 2 of 40 (40 Packets)

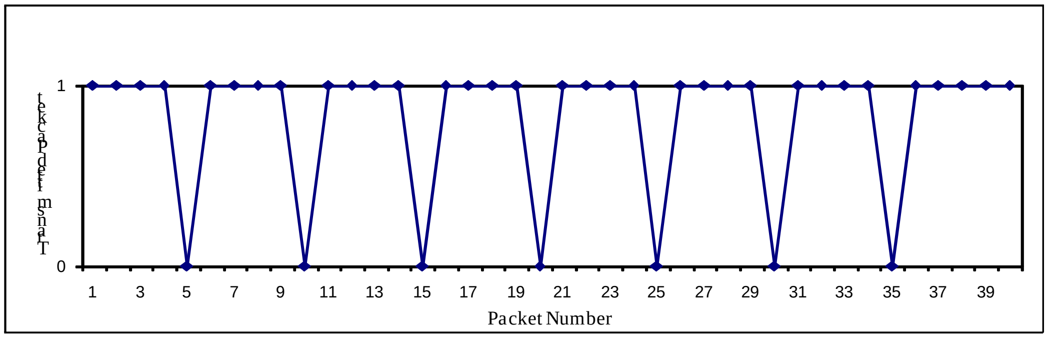

Fig 6.1

Fixed Position Packet Loss - run 3 of 40 (40 Packets)

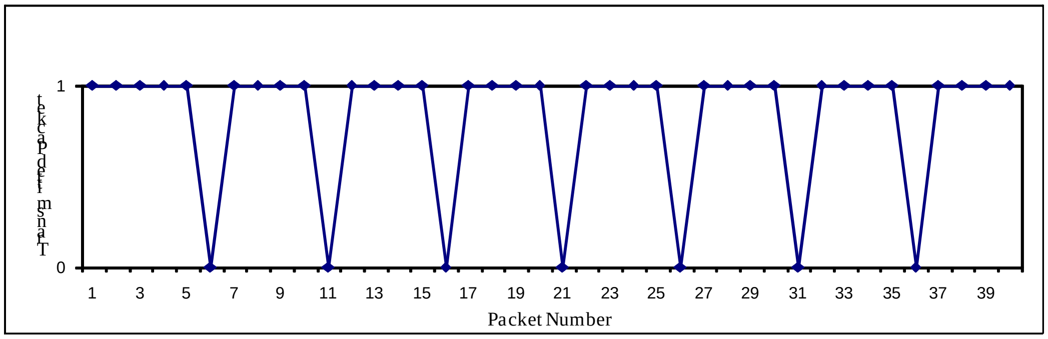

Fig 6.2

The pattern behaves as expected for first two repetitions and insert packet losses at specified positions as shown in fig 6.1. From third run the behavior of pattern changes, packets at specific positions are not dropped, the first packet loss is shifted to next position and this change is applied to all packet loss positions. In fig 6.2 packet losses were expected to be inserted at position 5, 10, 15, 20, 25, 30 and 35, but packet losses are inserted at position 6, 11, 16, 21, 26, 31 and 36. This behavior continues for every experimental run and packet loss position is incremented by one.

Overall behavior- Fixed Position Packet Loss (40 Packets)

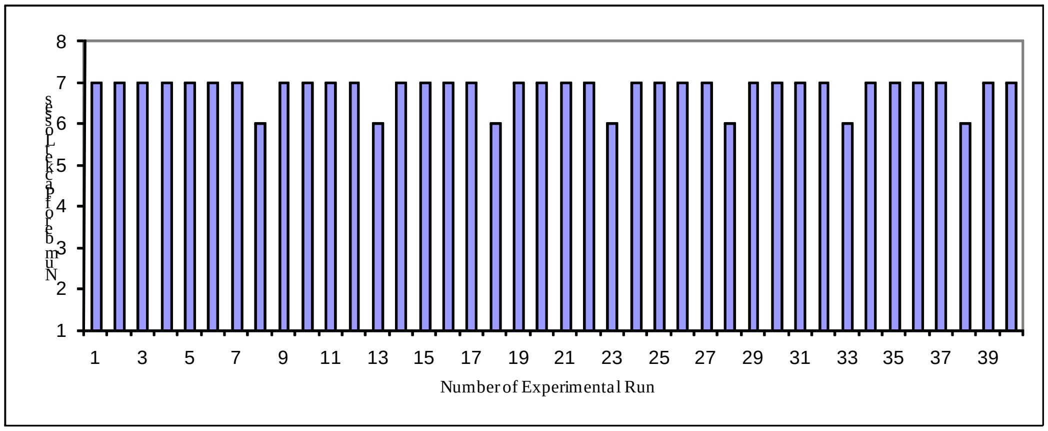

Fig 6.3

Fig 6.3 shows the overall behavior of pattern for 40 experimental runs. There are exact number of packet losses as specified in pattern i.e. 7 for most of experiments and 6 packets losses instead of 7 in run number 8,13,18,23,28,33 and 38 as there is a shift in packet loss position for every run after first two runs .

ii) Number of packets greater than pattern size (80/40)

Fixed Position Packet Loss - run1 of 40 (80 Packets)

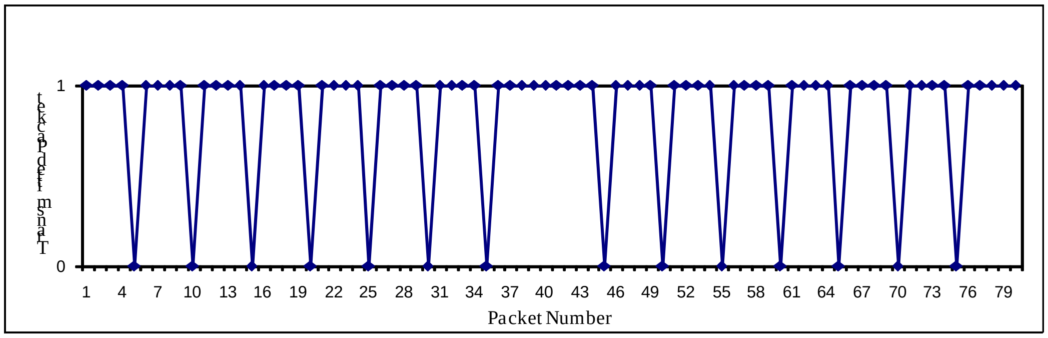

Fig 6.4

Fixed Position Packet Loss - run 2 of 40 (80 Packets)

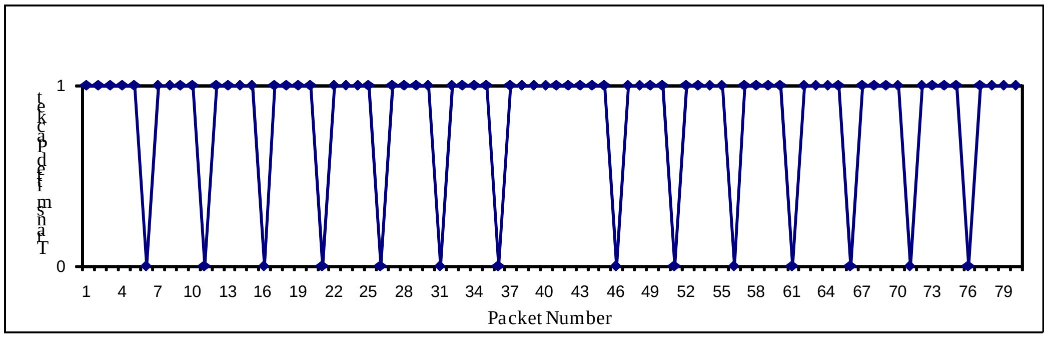

Fig 6.5

The pattern behaves as expected for first run (fig 6.4) and repeats itself twice in every run, as pattern size is 40 packets and number of packets sent is 80. From second experimental run (fig 6.5) the behavior of pattern changes, the first packet loss is shifted to next position and this change is applied to all the packet loss positions. This behavior continues for every experimental run and the packet loss position is incremented by one.

Overall behavior- Fixed Position Packet Loss (80 Packets)

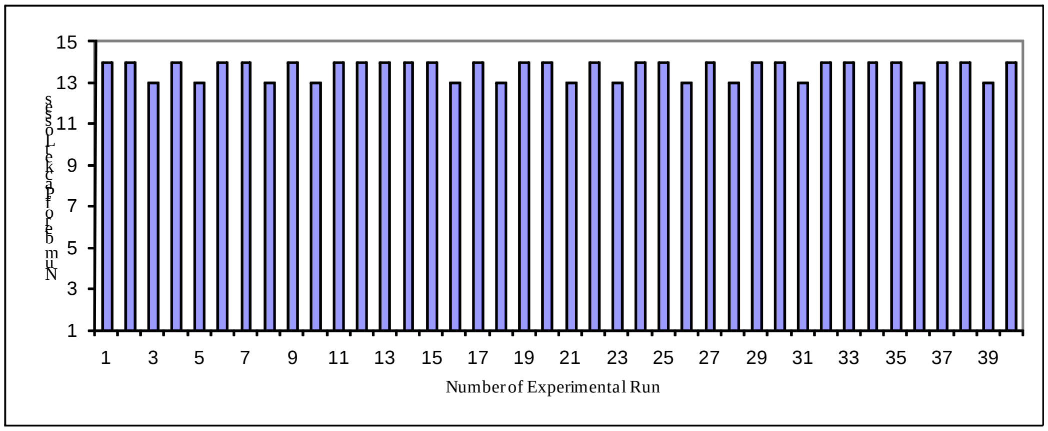

Fig 6.6

Fig 6.6 shows the overall behavior of pattern for 80 experimental runs. There are exact number of packet losses as specified in pattern i.e. 14 for most of experimental runs and 13 packets losses instead of 14 in run number 3, 5, 8, 10, 16, 18, 21, 23, 26, 28, 31, 36 and 39 as there is a shift in packet loss position for every run after first run.

6.1.2 Fixed Interval Packet Loss

Fixed interval patterns are defined to insert packet losses between fixed positions. Following pattern is created to insert packet losses at 1–4, 12–15, 25–28 and 33–37 position intervals.

```
./patt_gen -pkt -int test2.plp data 40 1,4,12,15,25,28,33,37

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config pattern test2.plp
```

´

i) Number of packets equal to pattern size (40/40)

Fixed Interval Packet Loss - runs 1, 2 of 40 (40 Packets)

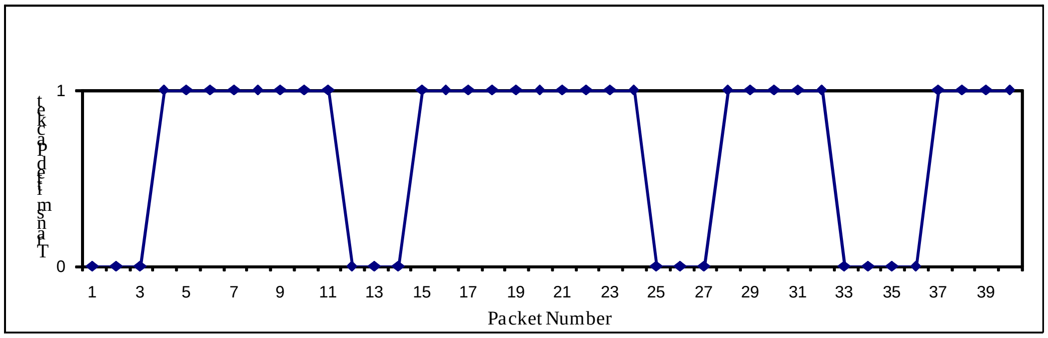

Fig 6.7

Fixed Interval Packet Loss - run 3 of 40 (40 Packets)

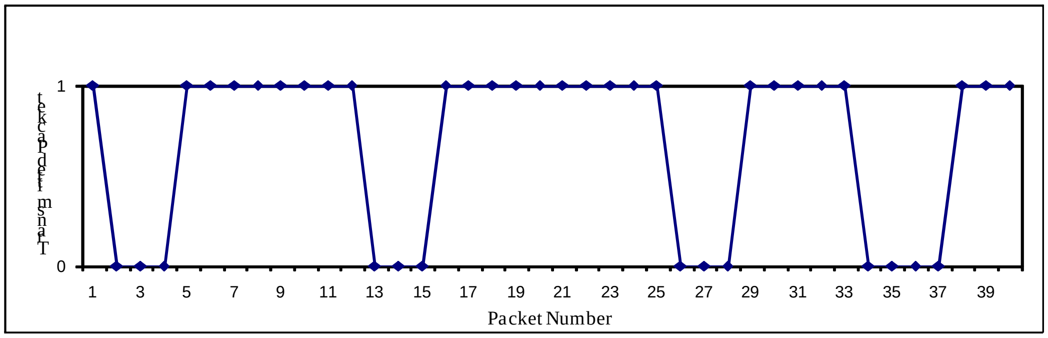

Fig 6.8

The pattern behaves as expected for first two repetitions and insert packet losses at specified intervals as shown in fig 6.7. From third run the behavior of pattern changes, the packets at specific interval are not dropped, the first packet loss is shifted to next position and this change is applied to all the packet loss positions. In fig 6.8 packet losses were expected to be inserted between positions 1–4, 12–15 , 25–28, 33–37, instead packet losses are inserted at positions 2–5, 13–16,26–29 and 34–38. This behavior continues for every experimental run and packet loss position is incremented by one in all loss intervals.

Overall behavior – Fixed Interval Packet Loss (40 Packets)

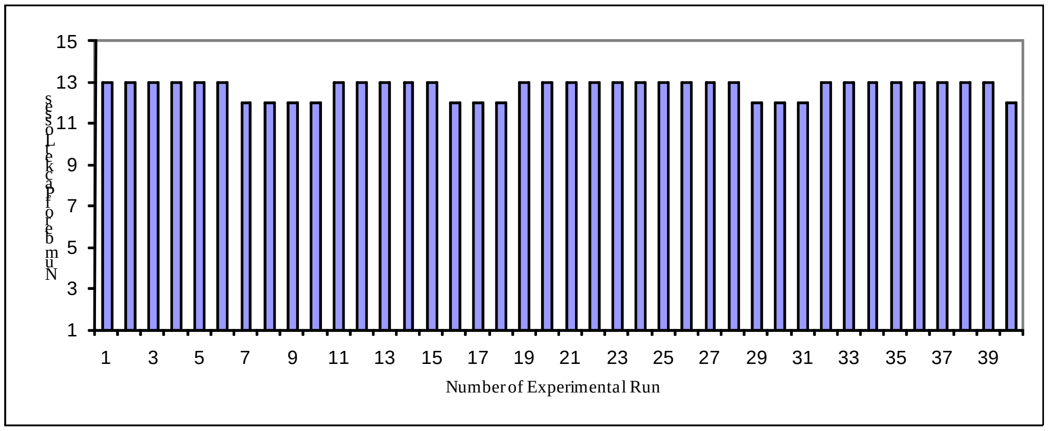

Fig 6.9

Fig 6.9 shows the overall behavior of pattern for 40 experimental runs. There are exact number of packet losses as specified in pattern i.e. 13 for most of experiment runs and 12 packets losses instead of 13 in run number 7,8,9,10,16,17,18,29,30,31 and 40 as there is a shift in packet loss position for every run after second run.

ii) Number of packets greater than pattern size(80/40)

Fixed Interval Packet Loss - run 1of 40 (80 Packets)

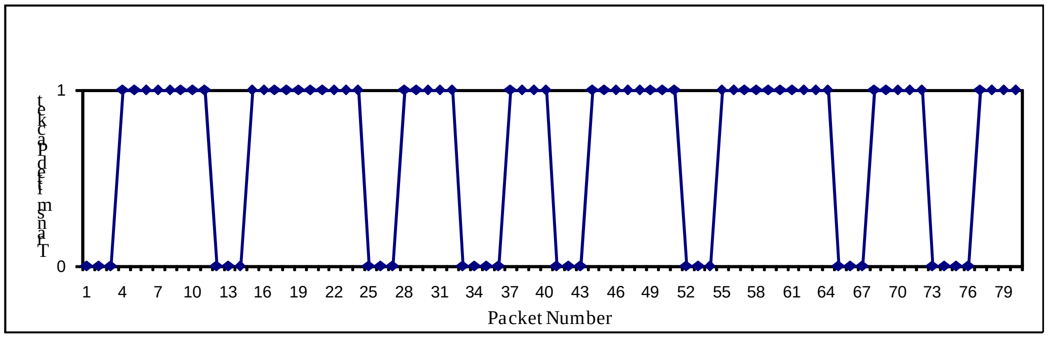

Fig 6.10

Fixed Interval Packet Loss - runs 2 of 40 (80 Packets)

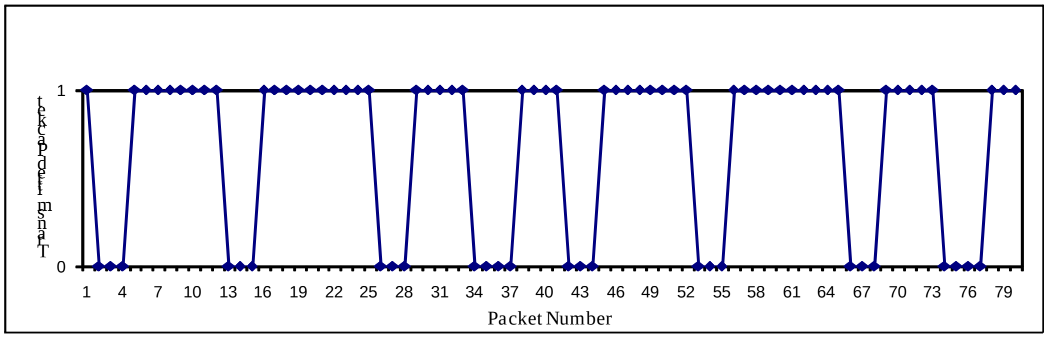

Fig 6.11

The pattern behaves as expected for first run (fig 6.10) and pattern repeats itself twice in every run, as the pattern size is 40 packets and the number of packets sent is 80. From second experimental run (fig 6.11) the behavior of pattern changes, the first packet loss is shifted to next position and this change is applied to all interval positions. This behavior continues for every experimental run, and the interval position is incremented by one.

Overall behavior – Fixed Interval Packet Loss (80 Packets)

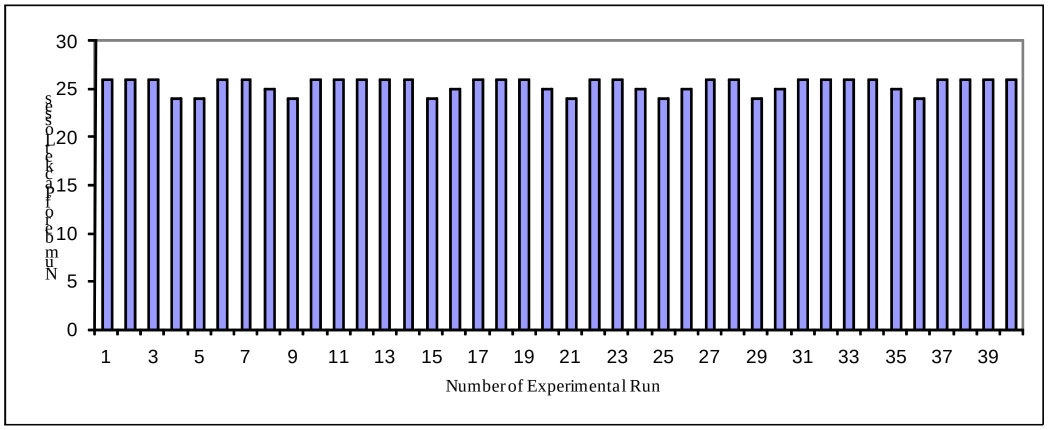

Fig 6.12

Fig 6.12 shows the overall behavior of pattern for 40 experimental runs. There are exact number of packet losses as specified in pattern i.e. 26 for most of experimental runs and 25 packets losses instead of 26 in runs 8,16,20,24,26,30 and 35.There are 24 packets losses in experiments 4,5,9,15,21,25,29 and 36.

6.1.3 Random Uniform Packet Loss

Uniformly distributed random packet losses can be inserted by specifying packet loss ratio. If value of PLR is less than 1, there will be approximate uniform packets losses depending upon the value of PLR. If value of PLR is greater than one, there will be exact number of packet losses equal to PLR value at random positions.

i) PLR < 1

Pattern is created with 0.3 PLR value to insert 30 % uniformly distributed random packet losses. Specific seed (654320) is used for uniform distribution.

```
./patt_gen -pkt -rand test3.plp data 40 654320 0.3

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config pattern test3.plp
```

a) Number of packets equal to pattern size (40/40)

Overall behavior – Random Uniform Packet Loss PLR<1 (40 Packets)

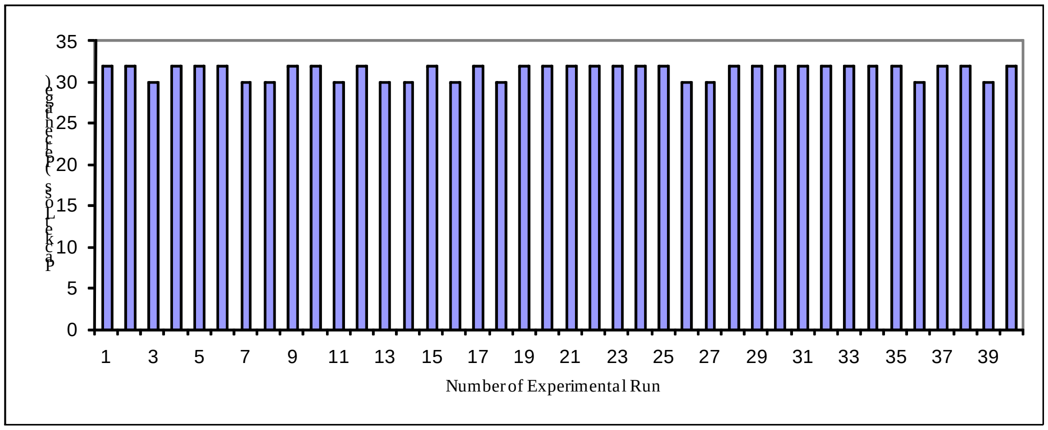

Fig 6.13

Fig 6.13 shows that packet losses are between 30 % and 32 % in all 40 experimental runs which shows the behavior of pattern as expected with slight variations

b) Number of packets greater than pattern size (80/40)

Overall behavior – Random Uniform Packet Loss PLR<1 (80 Packets)

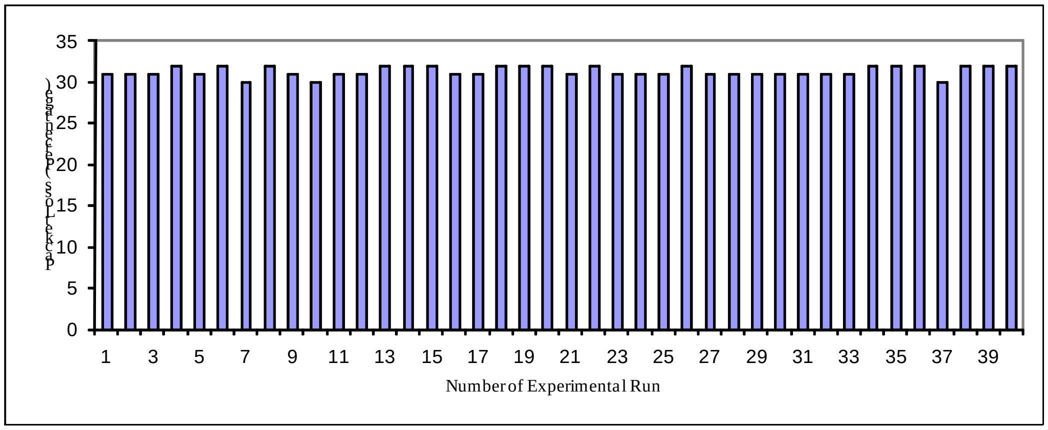

Fig 6.14

Fig 6.14 shows that packet losses are between 30 % and 32 % in all 40 experimental runs which shows the behavior of pattern as expected with little variations.

ii) PLR > 1

Following pattern is created with PLR equal to 7 i.e. there should be exactly 7 packet losses inserted by the pattern.

```
./patt_gen -pkt -rand test4.plp data 40 654320 7

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config pattern test4.plp
```

a) Number of packets equal to pattern size (40/40)

Overall behavior – Random Uniform Packet Loss PLR>1 (40 Packets)

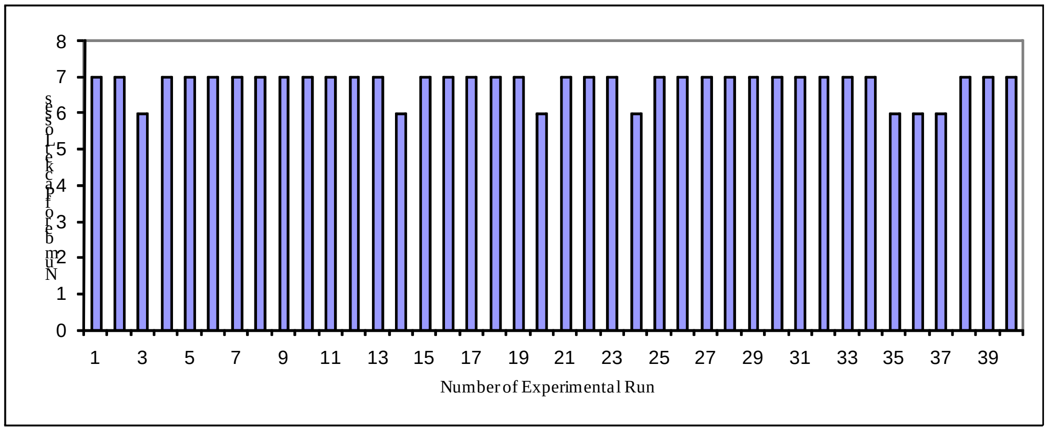

Fig 6.15

Fig 6.15 shows the overall behavior of pattern for 40 experimental runs. There are exact number of packet losses as specified in pattern i.e. 7 for most experimental runs and 6 packets losses instead of 7 in run number 3, 14, 20,24,35,36 and 37, as there is unexpected behavior of the pattern in some case.

b) Number of packets greater than pattern size (80/40)

Overall behavior – Random Uniform Packet Loss PLR>1

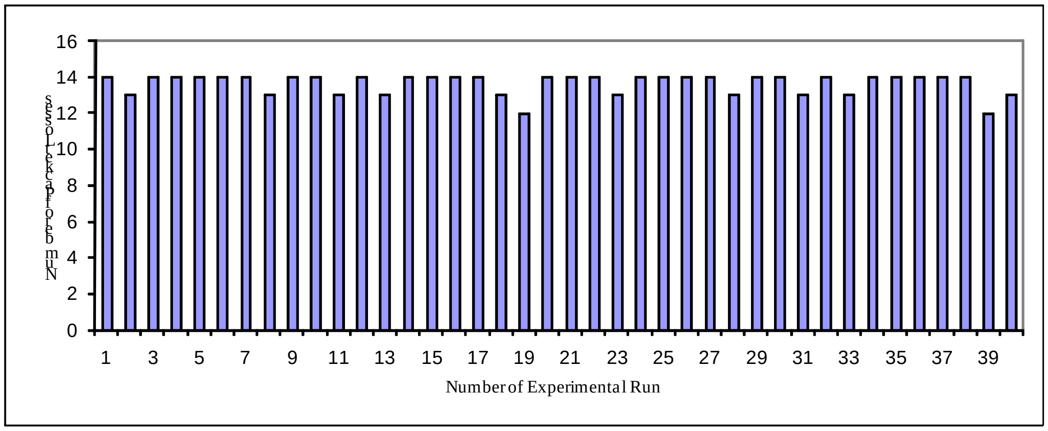

Fig 6.16

Fig 6.16 shows the overall behavior of pattern for 40 experimental runs. There is exact number of packet losses as specified in the pattern, i.e. 14 for most of experiment runs and 13 packets losses instead of 14 in run number 2, 8, 11, 13, 18, 23, 28, 31, 33 and 40 and 12 packet losses in run number 19 and 39 as there is unexpected behavior of the pattern in some case.

6.1.4 Gilbert-Elliot Packet Loss

Gilbert-Elliot model can be applied on patterns to insert packet losses according to the model. Model has two states, a good state and a bad state. PLR for each of these states can be defined along with transition probability to enter from one sate to other state.

Following pattern is created with 0.5 good state PLR, 0.8 bad state PLR, 0.08 good transition probability and 0.15 bad transition probability with pattern size of 100 packets.

```
./patt_gen -pkt -ge test5.plp data 100 0 0.5 0.8 0.08 0.15

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config pattern test5.plp
```

Pattern behavior – Gilbert Elliot Model

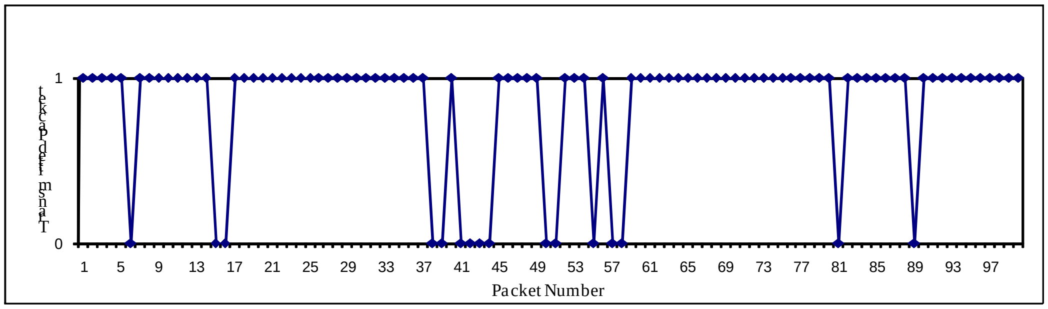

Fig 6.17

Fig 6.17 shows the behavior of pattern by inserting 16 packet losses at different positions according to well known Gilbert Elliot model.

6.2 Packet Loss Time-Driven Mode

6.2.1 Fixed Interval Packet Loss

In time-driven mode packet losses are inserted at specific time in milliseconds. Following pattern is generated to insert packet losses between 400 and 700 milliseconds, and the pattern is applied to 100 packets with 100 ms packet generation speed (inter packet arrival time).

```
./patt_gen -pkt -int test6.plp time 1000 400,700 0

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config pattern test6.plp
```

100 packets sent with 100 ms inter-packet time

Time-driven packet loss

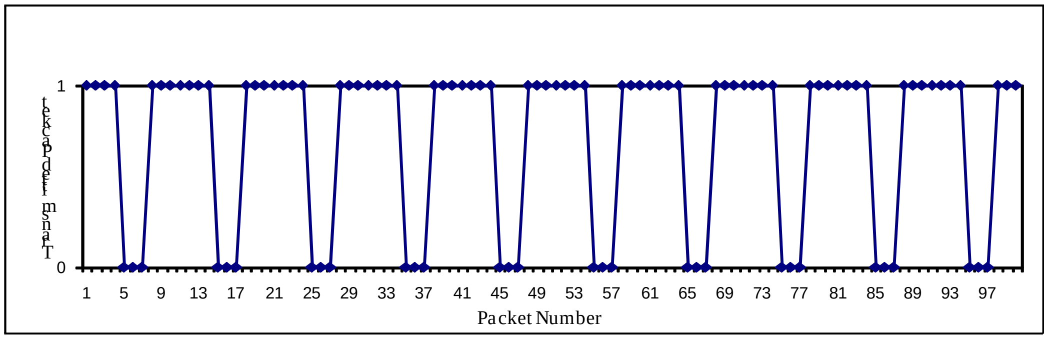

Fig 6.18

Fig 6.18 shows the behavior of pattern with 100 packets transmitted with inter-packet time of 100 ms and pattern size of 1000 ms. Then the pattern is repeated 10 times for 100 packets. In first repetition of pattern, packet number 4 is not affected and 5, 6 and 7 are dropped which come during the interval of 400 to 700 ms. This behavior of pattern continues till 100 packets are transmitted and all those packets are dropped that are transmitted between 400 and 700 ms interval during each repetition of pattern i.e no such problem as in data-driven case.

6.3 Delay Changes

Delay changes are applied to control and shape network traffic, different scenarios are created to check the behavior of KauNet with delay change patterns. Patterns are created to increase and decrease delays and checked with different inter-packet arrival times. Patterns have been applied to a given number of packets as specified in the pattern size, i.e. 40 packets. The experiment is repeated 40 times to check the behavior of patterns. Ping response time has been recorded and discussed with respect to delay changes with help of graphs.

6.3.1 Delay Increase (10 ms to 50 ms)

Pattern is created to apply 10 ms delay on first 10 packet and 50 ms to rest of packets to check the behavior of pattern with increase in delay.

```
./patt_gen -del -pos test1.dcp data 40 1,10,10,50

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config delay 10 ms bw 1 Mbit/s pattern test1.dcp
```

Delay increase (10 ms to 50 ms)

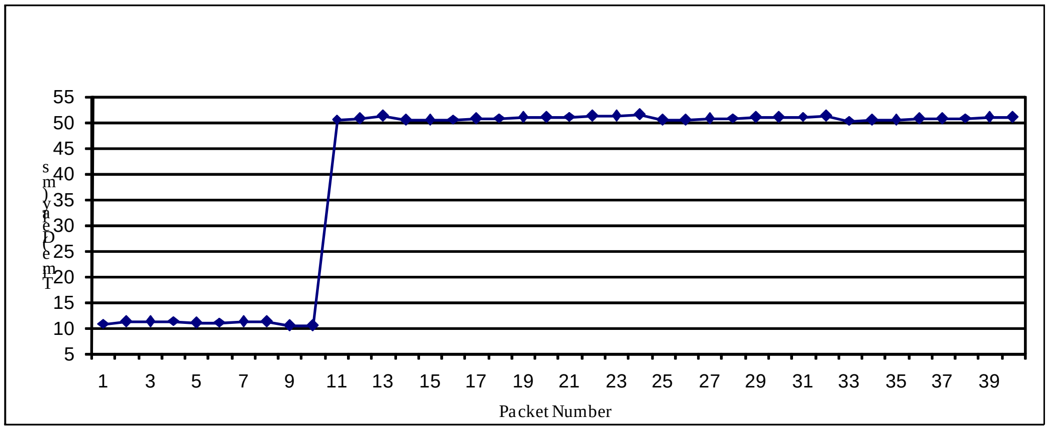

Fig 6.19

Fig6.19 shows the behavior of pattern with delay changes (increase in delay), 10 ms delay is applied on first 10 packets, there is sudden increase in delay of 50 ms as 11th packet arrives and it continues till last packet. In this case of delay changes, pattern behaves ideally as it is expected for first experimental run only. From 2nd run, it keeps applying 50 ms on all packets including first 10 packets which should experience 10 ms.

6.3.2 Delay Decrease (50 ms to 10 ms)

A pattern is created to apply 50 ms delay on the first 20 packets and 10 ms to rest of packets to check the behavior of pattern with a decrease in delay.

```
./patt_gen -del -pos test2.dcp data 40 1,50,20,10

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config delay 10 ms bw 1Mbit/s pattern test2.dcp
```

Delay decrease (50 ms to 10 ms)

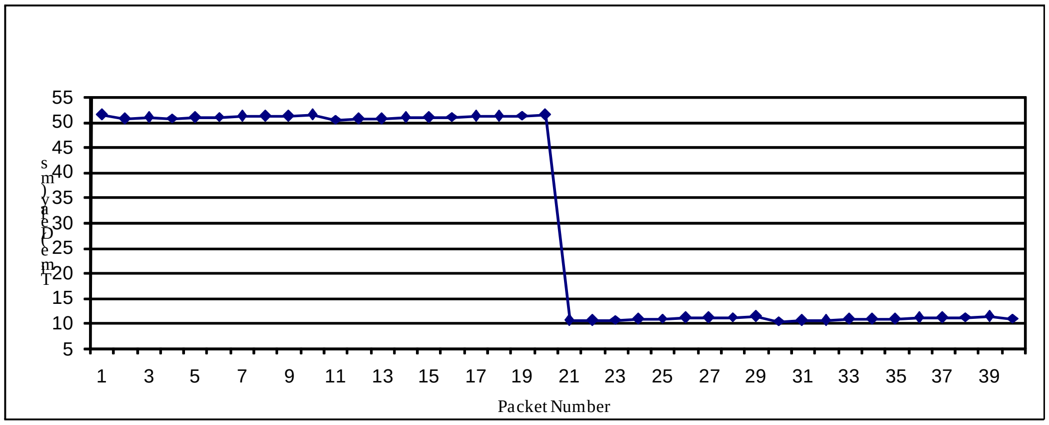

Fig 6.20

Fig6.20 shows the behavior of pattern with delay changes (decrease in delay), 50 ms delay is applied on first 20 packets, there is sudden decrease in delay to 10 ms when 21th packet arrives and it continues till last packet. In this case of delay changes, the pattern behaves ideally as it is expected for first experimental run only. From 2nd run it keeps applying 10 ms on all packets including first 20 packets which should be 50 ms.

6.3.3 Delay Decrease (500 ms to 50 ms with packet 100 ms inter-packet generation time)

Pattern is created to apply 500 ms delay on the first 20 packets and 50 ms to the rest of packets, every packet is generated after 100 ms to check the behavior of pattern with decrease in delay and handling of packets in queue.

```
./patt_gen -del -pos test3.dcp data 40 1,500,20,50

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config delay 10ms bw 1Mbit/s pattern test3.dcp

ping -c 40 -i 0.1 10.0.1.1
```

The option '-i 0.1' is used to set the inter-packet generation time equal to 100 ms.

Delay decrease (500 ms to 50 ms with 100 ms inter-packet generation time)

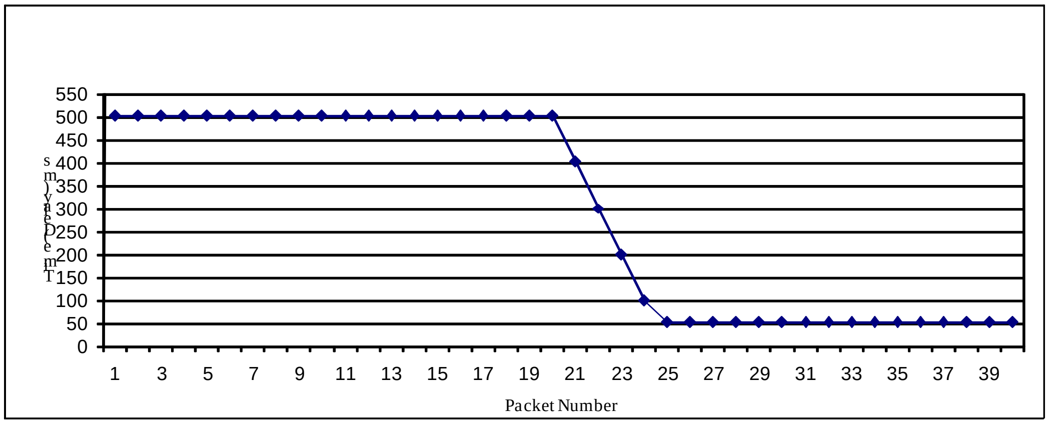

Fig 6.21

Fig 6.21 shows the behavior of pattern with delay changes (decrease in delay), 500 ms delay is applied on first 20 packets, and 21st packet is delay with 400 ms, 22nd with 300 ms, 23rd with 200, 24th with 100 ms and 25th with 50 ms instead of sudden decrease in delay to 50 ms at packet number 21 as in fig 6.20. Every packet is delayed with 500 ms up to packet number 20 and every packet is generated after 100 ms, there are 4 packets in queue after 20th.These packet are already delayed with 400,300,200 and 100 ms and are transmitted without further delay.

6.4 Bandwidth Changes

Bandwidth change patterns are created to increase and decrease bandwidth and check the behavior of patterns with different ICMP packet sizes. Patterns have been applied to exact number of packets as specified in pattern size, i.e. 40 packets. Experiments are repeated 40 times to check behavior of patterns. Ping response time has been recorded and discussed with respect to bandwidth changes with help of graphs.

6.4.1 Increase in Bandwidth

A pattern is created which limits the bandwidth to 100 kbps for the first ten packets and 1 Mbps for rest of packets. The impact of a pattern is checked by transmitting normal size of ICMP packets (64 B) and with increased packet size of 512 and 1460 B.

```
./patt_gen -bw -pos test1.bcp data 40 1,100,10,1000

# ipfw –f flush

# ipfw –f pipe flush

# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config delay 10 ms bw 10 Mbit/s pattern test1.dcp
```

i) Increase in bandwidth (ICMP packet size 64 B)

40 ICPM packets are transmitted with 64 B packet size for each packet using following command.

```
ping -c 40 -i 0.1 10.0.1.1
```

Bandwidth increase (100 Kbps to 1 Mbps with packet size of 64 B)

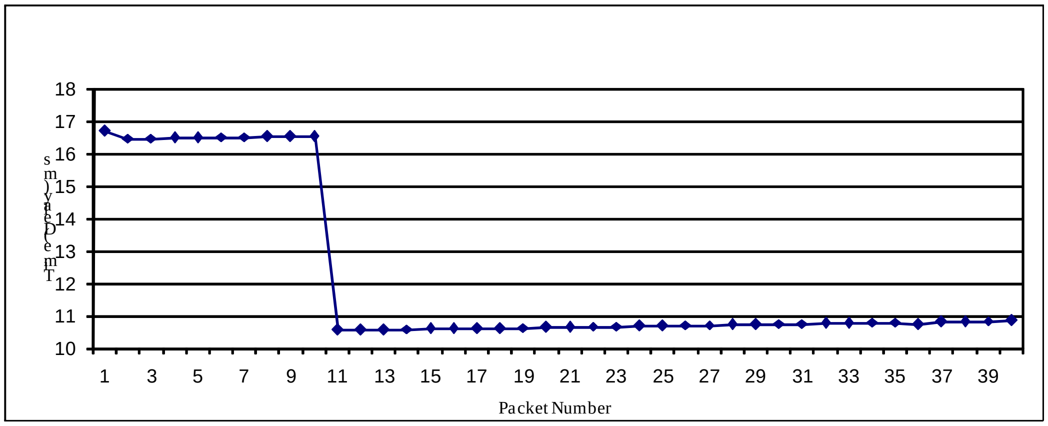

Fig 6.22

Fig 6.22 shows the impact of the pattern with bandwidth limitation of 100 kbps for first ten packets and then increased to 1Mbps from 11th packet onwards. Delay of 10 ms is applied to all packets along with bandwidth limitations by the pattern. In fig 6.22, the delay for first ten packets is about 16.5 ms, which is more than the applied delay of 10 ms. This increase in the delay is due to bandwidth limitations. From 11th packet there is change in bandwidth limitation, 1 Mbps bandwidth is applied on rest of packets which changes the delay from 16.5 ms to around 11 ms.

ii) Increase in bandwidth (ICMP packet size 512 B)

40 ICPM packets are transmitted with 512 B packet size for each packet using following command.

```
ping -s 512 -c 40 -i 0.1 10.0.1.1
```

Bandwidth increase (100 kbps to 1 Mbps with packet size of 512 B)

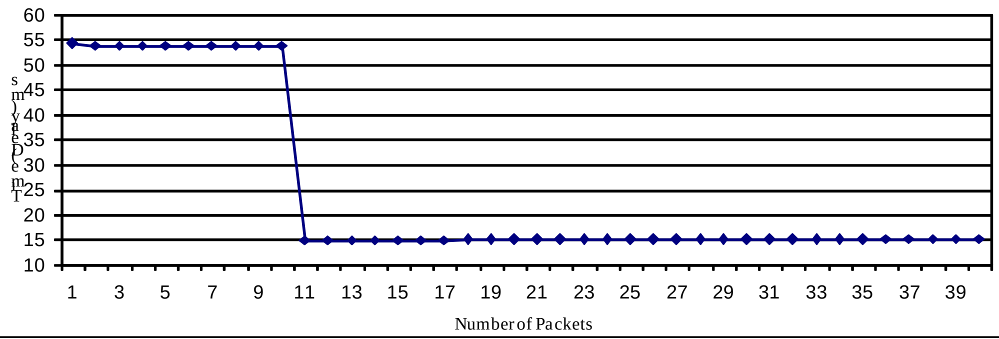

Fig 6.23

Fig 6.23 shows the impact of the pattern with bandwidth limitation of 100 kbps for first ten packets and it is increased to 1Mbps from 11th packet onwards with increased packet size of 512 B. In fig 6.23, delay for first ten packets is about 54 ms, which is more than the values shown in fig 6.22. Increase in packet size utilizes more bandwidth resulting large increases in delay. From 11th packet there is a change in bandwidth limitation, 1 Mbps bandwidth is applied on rest of packets that change delays to 15 ms. This delay value is more than that belonging to fig 6.22 due to the increase in packet size.

iii) Increase in bandwidth (ICMP packet size 1460 B)

40 ICPM packets are transmitted with 1460 B packet size for each packet using following command.

```
ping -s 1460 -c 40 -i 0.1 10.0.1.1
```

Bandwidth increase (100 Kbps to 1 Mbps with packet size of 1460 B)

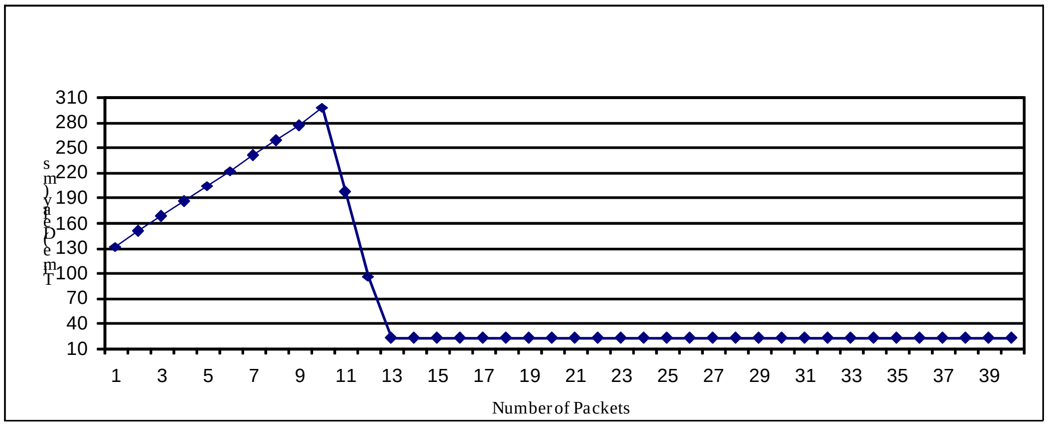

Fig 6.24

Fig 6.24 shows the impact of pattern with bandwidth limitation of 100 kbps for first ten packets and it is increased to 1 Mbps from 11th packet onwards with increased packet size of 1460 B. The delay is increased linearly and reaches up to 300 ms for 10th packet, as bandwidth is limited to 100 kbps. The availabe bandwidth is 100 kbps, whereas the required bandwidth is about 116.8 kbps, which adds extra delays on packets in the queue. From 11th packet, bandwidth is increased up to 1 Mbps which doesn’t suddenly reduce delay for packets 11 and 12 to 10 ms as specified in pattern because the packets 11 and 12 are already in the queue. It is about 200 ms and 90 ms for packets 11 and 12 respectively because these packets are already delayed due to small bandwidth. From the 13th packet, delay for every packet is about 21 ms, which is greater than the delays in case of 64 B and 512 B packet sizes.

6.4.2 Decrease in Bandwidth

A pattern is created which limits the bandwidth to 1Mbps for the first ten packets and 100 Kbps for the rest of packets. Pattern behavior is checked by transmitting ICMP packets of 64, 512 and 1460 B.

```
./patt_gen -bw -pos test2.bcp data 40 1,1000,11,100

# ipfw –f flush

# ipfw –f pipe flush
# ipfw add allow all from any to any

# ipfw add 1 pipe 100 icmp from 10.0.1.1 to 10.0.2.1 in

# ipfw  pipe 100 config delay 10 ms bw 10 Mbit/s pattern test2.dcp
```

i) Decrease in bandwidth (ICMP packet size 64 B)

40 ICPM packets are transmitted with 64 B packet size for each packet using the following command.

```
ping -c 40 -i 0.1 10.0.1.1
```

Bandwidth decrease (1 Mbps to 100 kbps with packet size of 64 B)

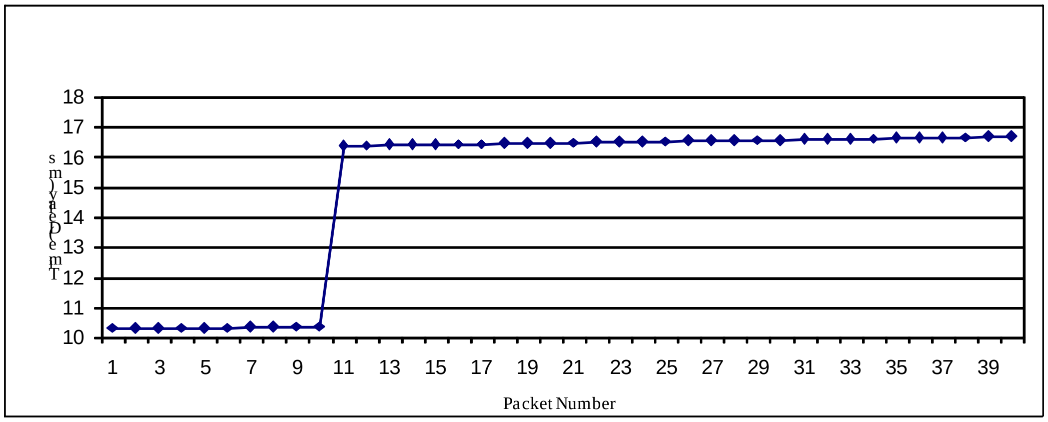

Fig 6.25

Fig 6.25 shows the impact of pattern with bandwidth limitation of 1 Mbps for the first ten packets and then decreased to 100 kbps from 11th packet onwards. In fig 6.25, delay for first ten packets is about 10.3 ms and from the 11th packet there is change in bandwidth limitation which changes the delay to 16.5 ms.

ii) Decrease in bandwidth (ICMP packet size 512 B)

40 ICPM packets are transmitted with 512 B packet size for each packet using the following command.

```
ping -s 512 -c 40 -i 0.1 10.0.1.1
```

Bandwidth decrease (1 Mbps to 100 kbps with packet size of 512 B)

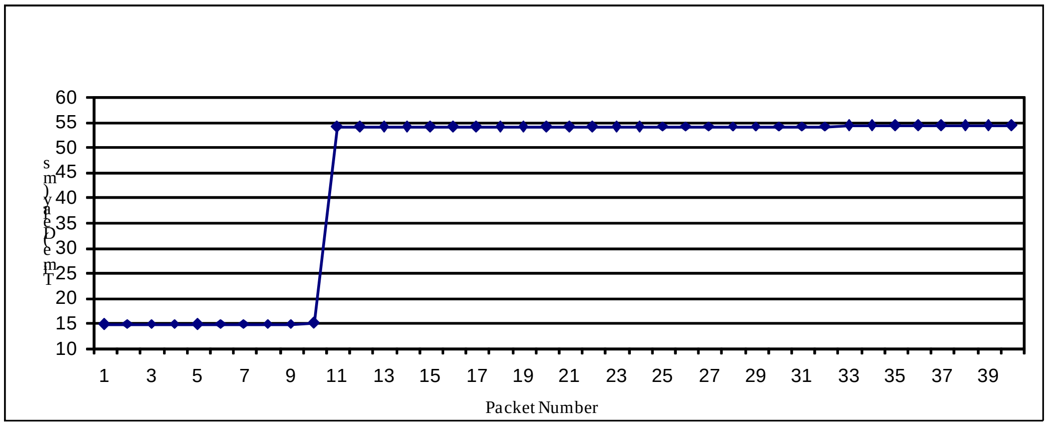

Fig 6.26

Fig 6.26 shows the impact of pattern with bandwidth limitation of 1 Mbps for first ten packets and then decreased to 100 kbps from 11th packet onwards with increased packet size of 512 B. In fig 6.26, delay for first ten packets is about 15 ms, which is more than in fig 6.25. From 11th packet 1 Mbps bandwidth is applied on rest of packets that increases delay to 54 ms because the packet size in this case is increased. Increase in packet size utilizes more bandwidth resulting increase in delay.

iii) Decrease in bandwidth (ICMP packet size 1460 B)

40 ICPM packets are transmitted with 64 B packet size for each packet using following command.

```
ping -s 1460 -c 40 -i 0.1 10.0.1.1
```

Bandwidth decrease (1 Mbps to 100 kbps with packet size 1460 B)

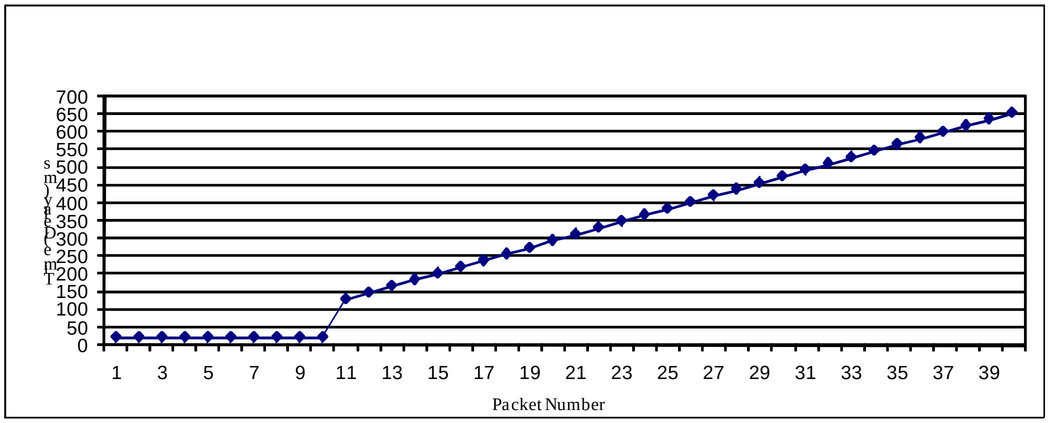

Fig 6.27

Fig 6.27 shows behavior of pattern with bandwidth limitation of 1 Mbps for first ten packets , followed by a decrease to 100 kbps from 11th packet onwards, and with increased packet size of 1460 B. Delay for first 10 packets is about 22 ms and from 11th packet delay starts increasing with a large factor as bandwidth is only 100 kbps and packet size is 1460 B, increase in packet loss and decrease in bandwidth adds delays to packets.

Chapter 7

HISTORY AND TRACE-BASED TRAFFIC SHAPING

7.1 History-Based Traffic Shaping

Traffic shaping is used to optimize and guarantee performance of a network. Uniformity in traffic flow and reproducibility in real network providing same delays and loss rates as in earlier times and maintaining the record of network traffic are driving forces towards the idea of history based traffic shaping. An observation period is used to collect information about delays applied on packets during transmission and same delays are used to apply on other sessions or traces. History of network traffic is maintained by collecting real network traces and extracting statistical information of traffic parameters like delay, jitter, loss rate and bandwidth utilization. These traffic parameters are saved and used to create patterns those are matched with future traces to perform traffic shaping introducing historical impact on network behavior.

7.2 KauNet as a History-Based Traffic Shaper

Measurement point, Database and a Pattern Generator Assistant are introduced to work along with pattern generator to apply history shaping mechanism on network traffic.

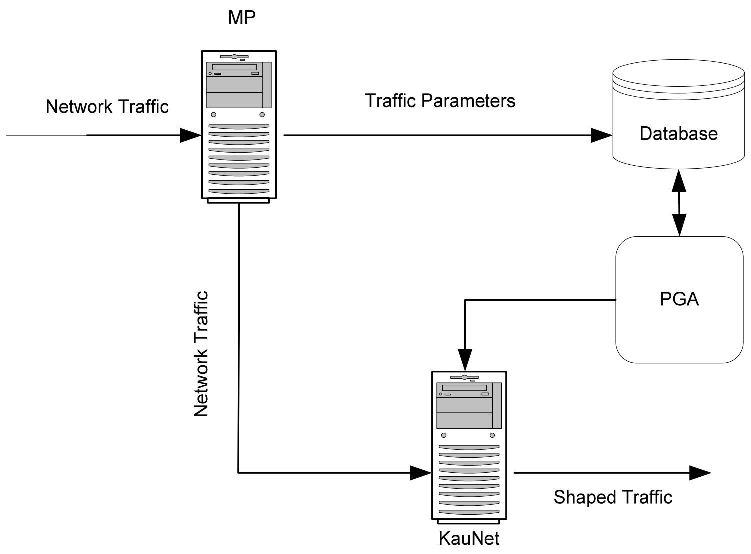

Fig 7.1

Traces are collected at measurement point and its information is stored into database to maintain history of network behavior. Information should contain packet sequence number, time stamp, delay on packets and utilized bandwidth.

The Packet Generator Assistant (PGA) should be able to pick information from database after specific time period on regular basis for various traces and perform statistical analysis (mean and standard deviation) of traffic delay, loss rate and bandwidth. The average values of a traffic parameters will ensure stable behavior while a change.

Required parameter values are passed to KauNet pattern generator to create pattern. This process is repeated continuously and patterns are generated after a specific interval. Pattern are dynamically loaded into kernel and applied on network traffic for shaping.

Before creating a pattern, the parameters of new pattern are compared with the parameters of loaded pattern. If the parameters of two patterns are same, there is no need to create the new pattern. New pattern should be created and loaded into the kernel only if the change in pattern remains within defined constraints of parameters.

Information about every pattern should be maintained in order to apply any previously generated patterns to respond sudden change in traffic or to create same behavior from history if needed.

7.3 KauNet as a Trace-Based Traffic Shaper

KauNet emulator uses pattern matching for traffic shaping, patterns can be created from real network traces, simulations, analytical expressions. Patterns are inserted into kernel or can be dynamically changed or switched by the use of command line tools to coop with dynamic event during emulation process.

Motivations behind trace based traffic shaping are to produce behavior of network traffic exactly as earlier traces. Some traces from real or emulated networks are collected (Ideal Traces) at a measurement point and analyzed to calculate exact values of traffic parameters e.g. bit errors, delay, packet losses and bandwidth utilization. KauNet patterns are created from mimicking the behavior of these ideal traces and patterns are inserted into the kernel to apply on emulated or real network traffic. In history-based traffic shaping information about network traffic is maintained where as in trace based traffic shaping ideal traces are used to form traffic patterns and apply on real or emulated traffic without maintaining the history of network traffic.

7.4 Limitations

KauNet applies packet losses, delay and bandwidth changes on the packets that enter KauNet emulator. If some packets are already dropped or delayed, KauNet cannot insert packets at packet loss positions and delayed packets are passed without decreasing delay. The solution to this problem is to use a perfect generator that generates packets without delays and losses.

Chapter 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this thesis work literature review of Quality of Service has been carried out and different models for Quality of Service have been discussed, most commonly used models are integrated and differentiated services model. Reactive and preventive traffic control has been discussed, and it has been stated how preventive traffic control can be applied to provide Quality of Service to end users by admission control, scheduling, policing and traffic shaping mechanisms. It is recommended to prefer preventive traffic control over reactive without affecting user perception to fulfill service level agreements. Different network emulators and shapers have been studied such as Linux Traffic Control, NIST Net, EMPOWER, Master Shaper, Easy Shape and DummyNet and discussed from a functional and usability perspective. DummyNet shaper can handle nondeterministic packet losses, delay and bandwidth changes using IPFW, a utility in FreeBSD. KauNet, an extension of DummyNet, provides deterministic bit errors, packet losses and exact control on delay and bandwidth changes. The tool pattern generator creates traffic patterns to match data packets for controlling network traffic parameters, and thus enables pattern-based traffic shaping. Experiments have been performed with different patterns for packet losses, delay and bandwidth changes. KauNet shows expected behavior for first few experimental runs to control packet losses, delay and bandwidth changes. However, the pattern is unable to repeat itself as expected after few experimental runs. In case of packets losses, pattern is repeated two times exactly as expected, but shows unexpected behavior for further repetitions. For delay and bandwidth changes, pattern acts ideally only for the first experimental run.

8.2 Future Work

The pattern generator is needed to be updated to solve pattern repetition issues as it shows unexpected behaviour after two repetitions in case of packet loss and after first repetition in case of delay and bandwidth changes. In case of Gilbert Eliot model, more experiments are needed to verify pattern behaviour for good and bad states with large network traffic. More experiments are needed to Evaluate KauNet with real time audio and video streaming data over larger networks. KauNet has been evaluated with TCP traffic, more experiments should be done with UDP and other protocols traffic. Patterns should be created by mimicking the real network traffic behaviour from collected network traces and apply these patterns to real network data creating uniformity and reproducibility in network traffic behaviour, introducing history based traffic shaping. To the date KauNet is compatible with FreeBSD 6.1 and FreeBSD 7.1, these compatibly issues should be solved for all version of FreeBSD. Furthermore, we recommend evaluation of the EMPOWER, as it provides policies for trace-based shaping.

REFERENCES

[1] White, P.P.; Crowcroft, J.,”The integrated services in the Internet: state of the art”, Proceedings of the IEEE, Volume 85, Issue 12, Dec 1997 Page(s):1934 – 1946

[2] Xipeng Xiao Ni, L.M., Michigan State Univ., East Lansing, MI; **“**Internet QoS: a big picture”, Network, IEEE, Mar/Apr 1999, Volume: 13, Issue 2, page(s): 8-18
ISSN: 0890-8044.

[3] Briane E.Carpenter, Kathleen Nichols, Senior Member IEEE, “Differentiated Services in the Internet”, Proceeding of the IEEE, Vol.90, No.9, September 2002.

[4] RFC 1190, http://tools.ietf.org/html/rfc1190.

[5] S. Avallone, G. Ventre, “A simple framework for QoS provisioning in traf.cengineered networks”, 2006 IEEE.

[6] Nordstrom, E.; Carlstrom, J.; Gallmo, O.; Asplund, L.,“Neural Networks for Preventive Traffic Control in Broadband ATM Networks” Communications Magazine, IEEE, Volume 33, Issue 10, Oct 1995

[7]Qin Zheng, “AN ENHANCED TIMED-ROUND-ROBIN TRAFFIC CONTROL SCHEME FOR ATM NETWORKS”,Local Computer Networks, 1996., Proceedings 21st IEEE Conference on Volume , Issue , 13-16 Oct 1996 Page(s):249 - 258.

[8] Andreas Brandt., Manfred Brandt, Stefan Rugel and Dietmar Weber,“Admission Control for Realtime Traffic: Improving Performance of Mobile Networks by Operating on Actual Throughput” , IEEE Communications Society / WCNC 2005 IEEE

[9] J. M. Peha, F. A. Tobagi, “Cost-Based Scheduling and Dropping Algorithms to Support Integrated Services”, IEEE Transactions on Communications, Vol. 44, No. 2, pages 192-202, February 1996

[10] M. Ritter, “Analysis of a Queuing Model with Delayed Feedback and its Application to the ABR Flow Control,” Report no. 164, University of Wurzburg Germany, Jan. 1997.

[11] Linux Advanced Routing & Traffic Control HOWTO

[12] http://pcquest.ciol.com/content/linux/2004/104040601.asp (2008-03-23)

[13] http://luxik.cdi.cz/~devik/qos/htb/ (2008-03-27)

[14] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. ACM Computer Communication Review, 27(1):31–41, January 1997

[15] M. Carson and D. Santay. NIST Net: A Linux-based network emulation tool. ACM IGCOMM Computer Communication Review, 33(3):111–126, 2003.

[16] P. Zheng and L. Ni. EMPOWER: A network emulator for wireline and wireless networks. In Proceedings of IEEE InfoCom. IEEE Computer and Communications Societies, March 2003.

[17] http://www.mastershaper.org (2008-04-13)

[18] http://luxik.cdi.cz/~devik/qos/imq.htm (2008-04-14)

[19] http://www.linuxbox.co.za/software/easyshape (2008-04-17)

[20] http://freshmeat.net/projects/htb.init (2008-04-19)

[21] http://sourceforge.net/projects/kaunet (2008-05-12)

[22] http://info.iet.unipi.it/~luigi/ip_dummynet/ (2008-05-15)

[23] http://www.freebsd-howto.com/HOWTO/Ipfw-HOWTO (2008-05-15)

[24] http://www.freebsd.org/doc/en/books/handbook/index.html (2008-05-16)