The University of Kansas

Information and Telecommunication Technology Center

Technical Report

Emulation of a Space Based Internet Communication Link: Design and Implementation

Sandhya Rallapalli
Gary Minden

ITTC-FY2003-24350-04

September 2002

Project Sponsor:
NASA
Glenn Research Center

Abstract

Space Based Internet (SBI) aims at networking in space. Apart from originating and terminating traffic, an SBI capable satellite shall handle routing issues in satellite system and relay traffic through a series of SBI satellites.

Space Based Internet emulation system aims at emulating networking in space. It evaluates the SBI system in an environment similar to a satellite communication system. The satellite data channels and the inter-satellite link properties are emulated in the system.

The thesis work presents the emulation design of the satellite communication interface and link properties. The architecture for Virtual Ethernet devices that emulate satellite interfaces is provided. The emulation of inter-satellite link features like bandwidth, propagation delay and bit error rates in the data transmissions forms the major part of the emulation. The creation of Virtual Ethernet devices and the emulation of communication link properties in Linux operating sytem provide challenges and extend the scope of Quality of Service in Linux.

This paper gives the design, implementation and results of satellite communication link emulation in Space Based Internet emulation system.

Table of Contents

Table of Figures

1 Introduction

1.1 Satellite Communications

Satellite Communications have developed considerably over the years to provide a mode of transmission for data including real-time audio, video signals and earth observation measurements. A satellite communication system is a potential medium for broadband networks due to wide area coverage, service consistency, on-demand bandwidth capability and ease of user scalability.

Earth Sciences Observation (ESO) satellites collect data related to land, marine systems, atmosphere and ecosystems. Data collected is minimally processed and stored on-board using high capacity data recorders. On coming in line-of-sight with a Geo-synchronous relay satellite like TDRSS, the stored data is transmitted. Relay satellites in space receive and downlink data to stations on the ground.

1.2 Networking in Space

In a networked system, ESO satellites route observation data efficiently to the receiving ground stations through a series of satellites. When satellites perform On-Board Processing to analyse the best routes to transfer data, networking in space is established. It provides alternative to high data rate, high capacity recorders on each satellite and eliminates the dependence on expensive communication services such as TDRSS.

An efficiently designed satellite communication system provides a transmission path from an ESO satellite at one end to a ground station at the other end of the earth. A constellation of satellites can be so designed that they provide continuous connectivity to ground stations all around the globe. An assortment of satellites from Low, Medium and Geo-stationary earth orbit groups can be used to create desirable satellite constellations depending on the requirements and cost factors.

1.3 Characteristics of Satellite Communication Links

The data communication links between satellites differ from terrestrial links in the sense of medium of transmission, propagation distances and atmospheric interference on the signals. Some of the characteristics of satellite communication links are discussed in the following sections.

1.3.1 Propagation Delay

The data transmitted by the sources in satellite communication system is subject to propagation delay. The main cause of propagation delay is the large distances in space compared to typical terrestrial links. The atmosphere also causes delay in data transmission. Large distances and other factors between a ground station and a GEO result in a typical Round Trip Time (RTT) of 260ms. Similarly, data transfer between a ground station and LEO results in RTT of 10-30ms and with MEO in about 100ms [8]. In the satellite communication system, maximum propagation delay occurs during data transfer between two GEO satellites resulting in about 550ms RTT.

Hence, propagation delay is one of the main characteristics under consideration in space communication.

1.3.2 Errors in Transmission

Data transmission between satellite communication elements is prone to errors due to various reasons [6]. Absorption in a medium is one of the reasons that cause loss of data. In the atmosphere, absorption can occur due to rain, fog, clouds, snow, hail, water, molecular oxygen and ionospheric effects. Dispersion, resulting due to the movement of different frequencies at different speeds along a channel, causes distortion of a signal. In addition to the losses due to the above factors, signal attenuation causes bit errors. A signal travelling between a ground station and a satellite attenuates in strength due to atmospheric gases, clouds and rainfall that absorb the energy from the wave. The inter-satellite signals are affected by free space loss, the loss in power of a signal as it travels through free space, resulting in errors in the signal. The bit errors result in data due to the combined effect of the medium of propagation, attenuation and atmospheric absorption in the transmitted signal.

1.3.3 Delay –Bandwidth Product

Due to the large propagation delay in satellite communications, the delay-bandwidth product is more compared to local terrestrial links. The delay-bandwidth product results in an increase in TCP window in satellite applications to store in-flight data.

1.3.4 Noise

One of the characteristics of a signal is the signal-to-noise ratio. Noise can affect a signal in space due to space elements like sun, moon, galactic noise, cosmic noise and atmospheric noise [6]. The effect of noise is predominantly on signals lower than 1GHz frequency.

1.4 SBI Emulation of Communication Links

Space Based Internet (SBI) aims at networking between ESO satellites in space. SBI satellites are capable of forming a network in space to route data from ESO satellites to ground stations. An SBI satellite shall process data and routing information and no longer cache data on-board.

SBI intends to use Internet Protocol (IP) as the network layer protocol. IP is widely used as network layer protocol in terrestrial networking. Defining a different protocol for satellite communications using SBI leads to synchronization and updation issues with terrestrial networking.

An SBI system is illustrated in Figure 1. An ESO collects data from area A and transmits data to a relay satellite RS1. The relay satellite RS1 performs routing operations and decides whether to transmit the data to the next relay satellite RS2 or the ground station GS1. Thus, data from source S can reach destination D through a series of relay satellites (RS1, RS2, etc.,).

Space Based Internet emulation system emulates the SBI environment. It emulates the inter-satellite communication links and their Quality of Service attributes. The main attributes of communication links under consideration – bandwidth, propagation delay and bit error rates are emulated in the SBI emulation system. This work gives the details of the communication link emulation design and implementation.

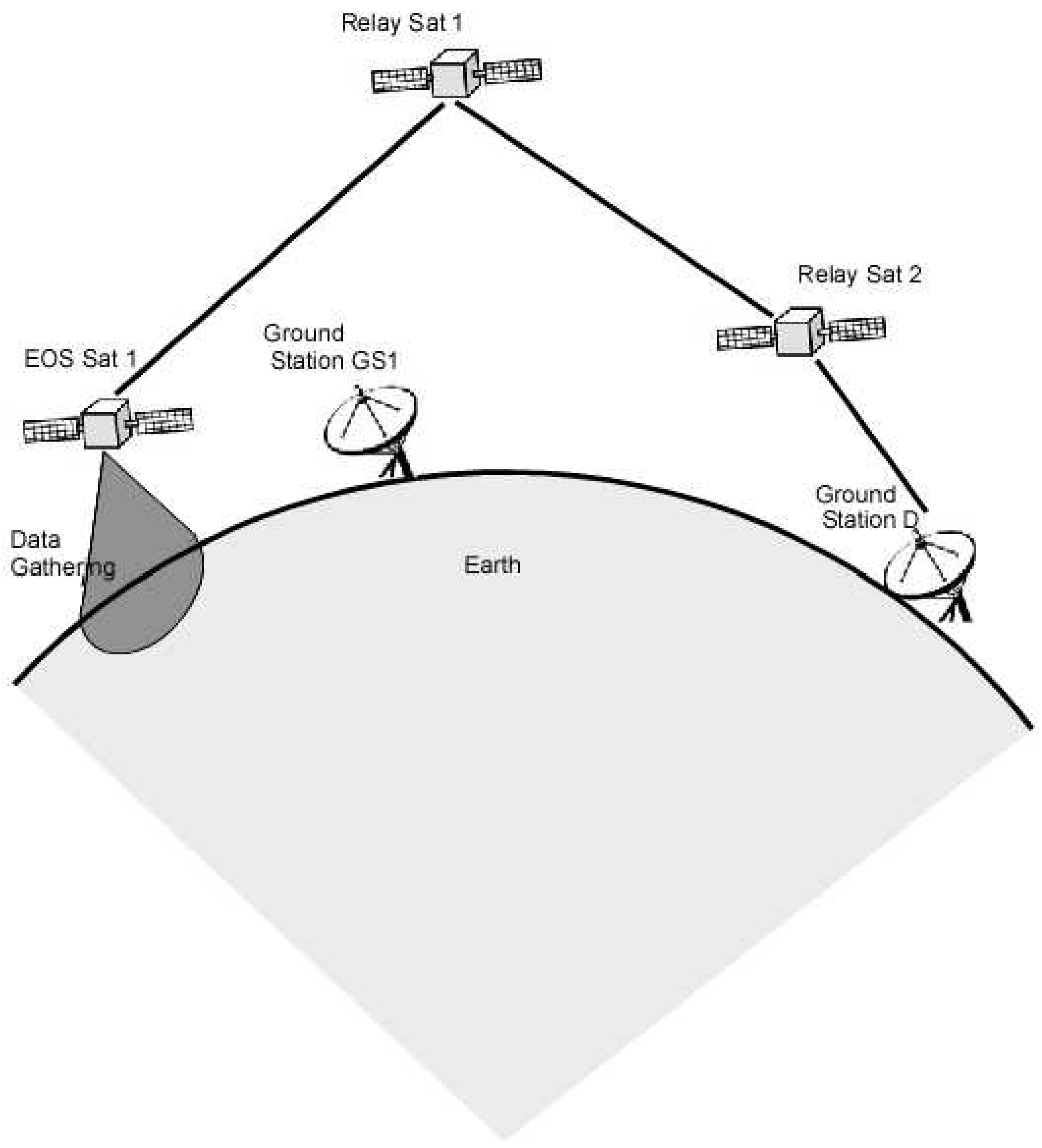


Figure 1: Illustration of Space Based Internet environment

1.5 Thesis Organization

The thesis presents the design and implementation of the emulation of a satellite communication link. The initial chapters explain the design for the emulation of Ethernet devices and simulation of QoS features. The later chapters explain the implementation of the modules described in the previous sections in the SBI emulation. The results for the implementation are provided in the final chapter.

The second chapter gives the SBI emulation system architecture, Linux QoS and the API for QoS in this chapter.

The third chapter explains the design and specifications of Ethernet device emulation.

The design for simulation of propagation delay in SBI emulation system is described in detail in the fourth chapter.

The fifth chapter explains the design of Bit Error Rate simulation in the SBI emulation system.

The sixth chapter gives a detailed description of incorporation of propagation delay and bit error rates attributes into a packet flow. It provides the QoS features for the Space Based Internet emulation.

Chapter seven describes the integration of the satellite communication link emulation and their QoS properties designed in the previous chapters in the SBI Emulation.

Chapter eight provides the testing scenarios, performance analysis and results for the work described in this document.

Chapter nine provides the conclusions derived and future work possible.

2 Structure of SBI Node

This chapter provides a background to the work presented in this document. An introduction to SBI emulation system is required to understand the emulation and assignment of QoS attributes to the communication links. The first section gives an introduction to the SBI emulation system. The second section gives a brief description of the Linux QoS architecture that provides a background for QoS designs provided in the later chapters. The third section gives an introduction to API for Linux QoS used in setting the communication link parameters in SBI emulation system.

2.1 SBI Emulation

Space Based Internet Emulation system aims at emulating networking in space. The architecture of the SBI emulation system is mainly divided into Emulation Manager and Node modules as shown in Figure 2.

The Emulation Manager is the centralized controller of the emulation system. It controls the emulation scenario operations, manages the flow of messages between the SBI Nodes via the Management Network and monitors the status of the emulation. The Emulation Manager also controls the Node communication parameters and the link properties in the emulation system.

The SBI Node emulates a satellite or a ground station in a space communication system. Each interface on the Node represents a communication channel on a satellite. The emulation software configures the interfaces and sets the communication properties like the propagation delay, bandwidth and bit error rate on each interface on the Node. Emulation Manager controls the communication link parameters.

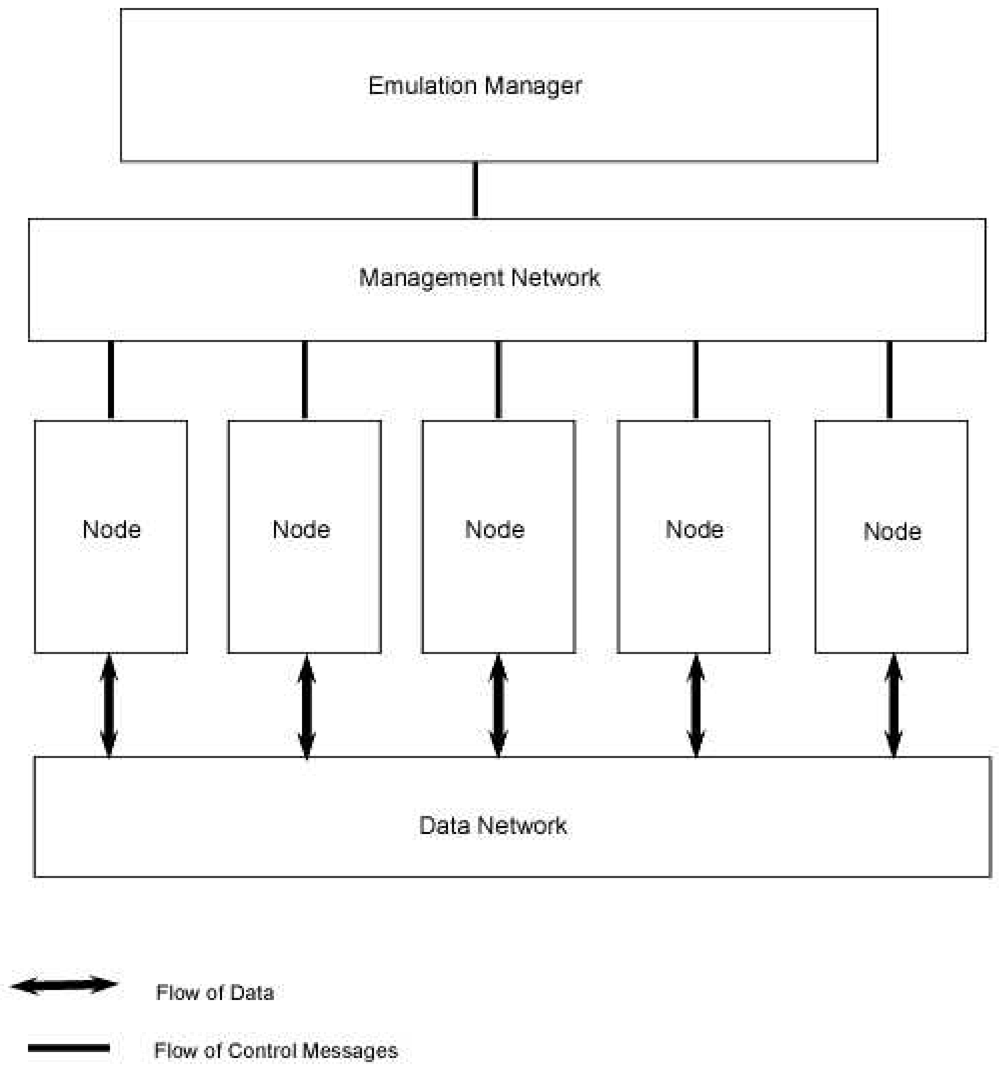


Figure 2: SBI Emulation System

Space Based Internet Emulation System architecture in detail can be referenced in [12]. The interaction between Emulation Manager and the Nodes in the SBI emulation system is shown in Figure 2.

The interaction between the Emulation Manager and the Node modules is performed using data and management networks. The Data Network is used for data transfer between the Nodes. The Management Network is used to transmit control messages between the Emulation Manager and the Nodes.

The architecture of SBI Node is given in detail in the following section.

2.1.1 SBI Node

The Space Based Internet Node emulates Earth Sciences Observation (ESO) satellite in the satellite communication system. Each antenna on a satellite is represented by an interface on the Node. The number of interfaces used on the Node in the emulation system represents the number of antennae on a satellite. An SBI Node must be capable of originating, terminating, relaying and providing Quality of Service to data from other nodes. The architecture for SBI Node is shown in Figure 3.

2.1.1.1 Communications Module

The Communications Module processes the messages passed by EM and passes control to the Instrument Scheduling and Routing modules for further processing.

2.1.1.1.1 Instrument Scheduling

The Instrument Scheduling module receives messages from the Communications Module to schedule data transmissions from the Node. A data generator is used to originate traffic from the Node. In SBI emulation, data is generated using the generator 'Netspec'.

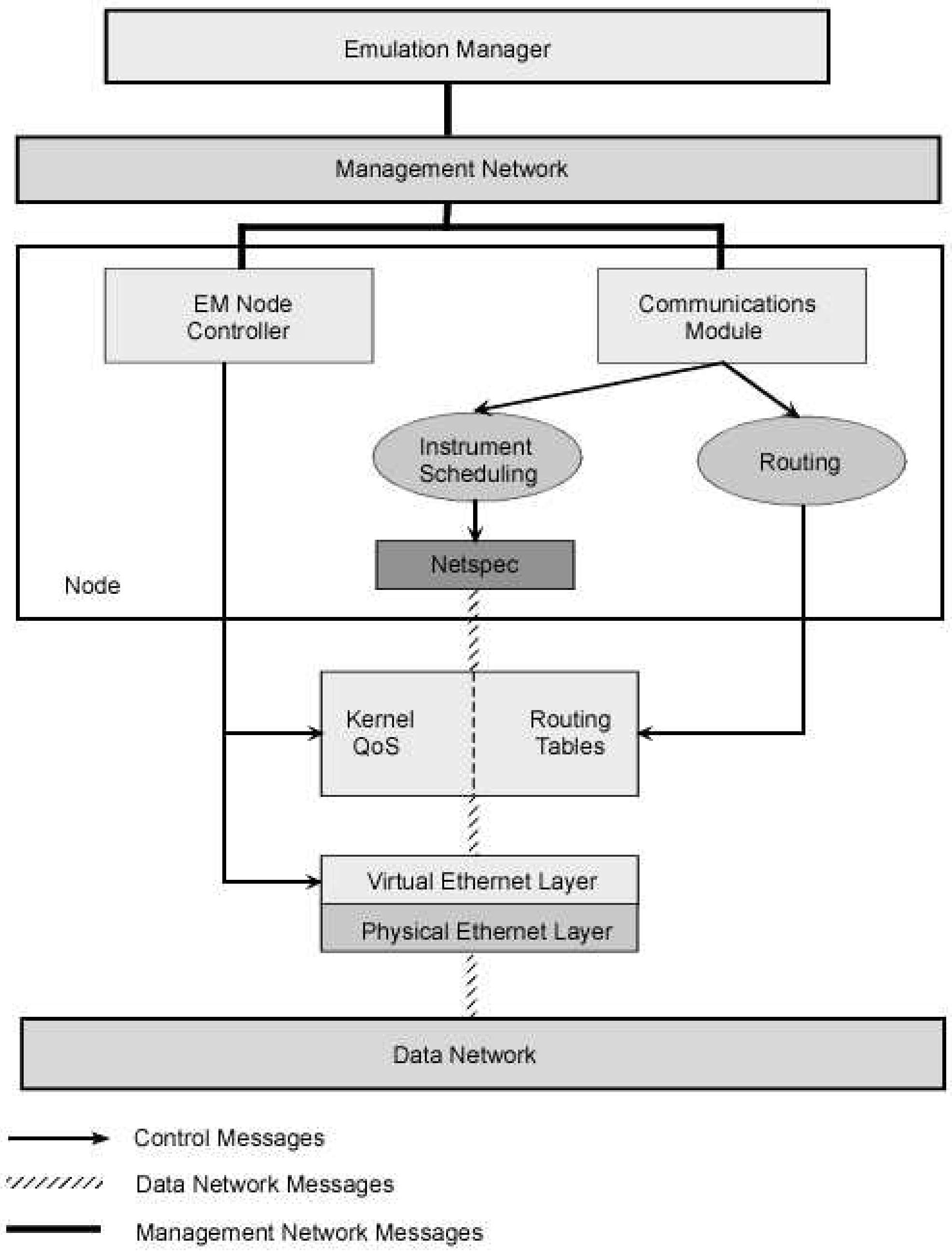


Figure 3: Space Based Internet Node Architecture

2.1.1.1.2 Routing

The routing module proceses messages obtained from Communications Module. It directly controls the routing tables on the Node.

2.1.1.2 EM Node Controller

The EM Node Controller processes control messages passed from the Emulation Manager to the Node. It manages communication interfaces and their properties based on requests from the Emulation Manager. On receiving messages to setup or modify interface parameters, it contacts corresponding modules.

2.1.1.2.1 Interface

SBI emulates satellites and the communication links on the satellites used for data transfer. A typical satellite in space has a number of communication links with varying bandwidths installed to provide multiple data transfer channels.

In the SBI emulation system, a physical node emulates a satellite in space. An Ethernet device on the node capable of transmission emulates a communication link on a satellite. The concept of Virtual Ethernet devices has emerged to avoid the need to install more than one Physical Ethernet device on a node for emulation purposes. The concept is cost-effective and increases the scope of the number of communication channels between two nodes than the possible number of Ethernet cards on the nodes. Operations related to emulation interfaces are performed by Virtual Ethernet device layer in the Linux kernel.

2.1.1.2.2 Interface Properties

The features of satellite communication links – bandwidth constraint, propagation delay and bit error rate are emulated in the SBI emulation system. The properties are introduced into emulation packet flows using the concept of queuing discipline in Linux QoS. An introduction of Linux QoS and details required for design that follows in the later chapters is provided in the next section.

On each Ethernet device in Linux kernel, the properties of a single queuing discipline can be applied to packet flow through the device. Bandwidth control is applied using Token Bucket Filter (TBF) queuing discipline that is part of Linux QoS. Propagation delay is applied to a packet flow using Delay queuing discipline and bit error rate using BER queuing discipline. To merge the properties of the three queuing disciplines and apply to a packet flow, a new queuing discipline ‘serialq’ is designed. Serialq concatenates the TBF, Delay and BER queuing disciplines in serial order. The format of serialq queuing discipline is shown in Figure 4

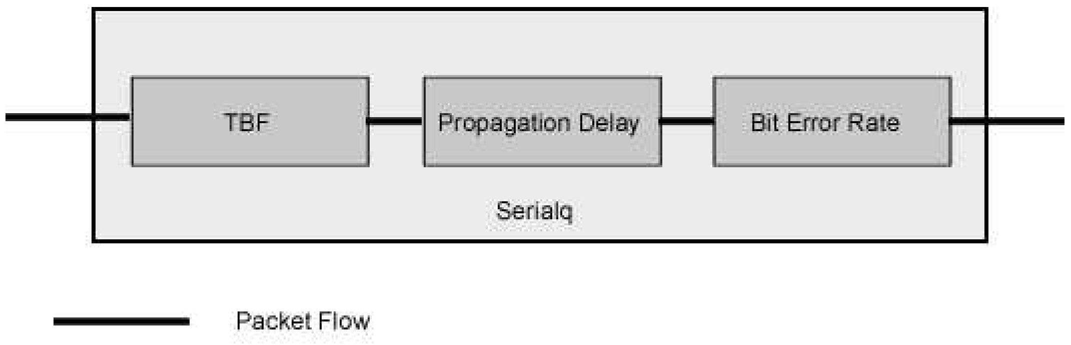


Figure 4: Format of Serialq Queuing Discipline

2.2 Linux QoS

Linux traffic control offers Quality of Service to packet transmissions through interfaces on a node. The traffic control system is implemented using queuing disciplines, classes and filters. The outgoing packets are queued in Queuing disciplines (qdiscs) before trasmitting through an interface. A Class contains a qdisc, by default a First-In First-Out (FIFO) qdisc. A Filter is attached to a class and is used to determine whether a packet belongs to the class based on certain characteristics. The details of Linux QoS are available at [18].

2.2.1 Queuing disciplines

Linux QoS uses qdiscs to queue packets set for transmission on an interface. Each qdisc has unique properties that are applied to the packets in it's queue. Linux QoS provides a number of qdiscs including First-In First-Out (default), Class Based Queuing (CBQ), Token Bucket Filter (TBF), Random Early Detection, Stochastic Filter Queuing.

Each qdisc has an element of format 's*truct Qdisc*' corresponding to it. This contains the pointer to specific qdisc parameters, packet queue and other qdisc generic information. Each qdisc can have a local structure to store specific information. This information can be accessed through *data* pointer in '*struct Qdisc*'. Also, attached to a qdisc is an element of format '*struct Qdisc_ops*'. The element contains the routines used to initialize and destroy qdisc, enqueue, requeue and dequeue packets, modify and list qdisc attributes.

2.2.2 *Class Based Queuing*

A qdisc can be class-based in the sense that a qdisc can contain classes inside it. CBQ is one of the qdiscs that contain classes and qdiscs. This capacity of CBQ allows the creation of a complex setting for implementing QoS on a packet flow. Filters are used to separate packet flows into classes based on various features like the source, destination IP address, source, destination port and Type of Service.

The process of setting up CBQ and it's classes and qdiscs is shown:

- Create CBQ on the interface on which it is desired.
- Create the required number of classes inside the CBQ
- Create qdiscs on each class inside the CBQ. The qdiscs can be any one of the qdiscs provided in Linux QoS including CBQ itself.
- Create a filter for each class as required to separate the packets from the original packet flow into each class. In case of an absence of a filter on a class, any packet can enter the class.

This section gives the details of Linux QoS. The queuing disciplines for link properties like bandwidth, delay and ber are designed based on the Linux QoS principles. Once the QoS modules are created, the emulaton system needs to access them to use the qdiscs for the emulation. The interfaces provided for Linux QoS modules are described in the following section.

2.3 Interfaces for Linux QoS

Linux QoS modules can be accessed using a user interface ‘Iproute’ or an application programming interface ‘API for Linux QoS’.

2.3.1 Iproute

‘Iproute’ application provides a user interface for the Linux QoS. The qdiscs, classes and filters can be created, destroyed and managed using the command ‘tc’ provided by Iproute. The parameters given using ‘tc’ are passed into the kernel using netlink sockets where further processing is done in Linux QoS modules.

The command used to create an element – qdisc, class or filter is:

```
tc <element>  add  dev <device_name>  <handle_details>
<element type>  <element_details>
```

- *element* is either a qdisc, class or filter.
- *device_name* is the device on which the element is created.
- *handle_details* are the parent and handle information. More details can be found at [20].
- *element_type* is the type of element to be created. Ex: If a qdisc creation is requested, cbq can be element_type.
- *element_details* are the details required for each element type. This information can be found for any element_type by the command:

  ```
  tc <element>  add  <element_type>  help
  ```

Henceforth in this document, `dev <device_name> <handle_details>` are referred to as `<DEV_HANDLE>`.

Similarly, the command used to destroy an element is:

```
tc <element> del <DEV_HANDLE> <element type>
<element_details>
```

Using 'tc', modifications can be made to QoS elements. The command is:

```
tc <element> change <DEV_HANDLE> <element type>
<element_details>
```

The use of Iproute application in accessing Linux QoS modules is shown in this section. The delay, ber and serialq queuing disciplines are so designed that they can be managed from user space through Iproute interface.

2.3.2 API for Linux QoS

The Application Programming Interface (API) for QoS provides a programming interface that allows dynamic configuration of QoS attributes on an interface. The required parameters for the creation, modification or deletion of a QoS element – either a qdisc, class or filter are filled in defined formats and passed into the kernel QoS modules using rtnetlink sockets. The documentation on API for QoS can be found at [19].

The API for QoS is used to set the QoS attributes like bandwidth, propagation delay and bit error rate on SBI node interfaces. The procedure to use the API for TBF qdisc is described below. Any element of Linux QoS is similar to the following procedure.

An element of type '*struct qdisc_gen*' is filled with required parameters. The *qdisc_options* element points to the features of the queuing discipline desired for creation, modification or deletion. The format of '*struct qdisc_gen*' is shown:

```
struct qdisc_gen
{
      char dev[16];          /* Device */
      __u32 parent;          /* parent info */
      __u32 handle;          /* handle info */
      char qdisc_type[16];   /* Queuing Discipline type */
      void *qdisc_options;   /* common pointer for all queuing
                             discipline options */
};
```

As a sample, the layout of *tbf* queueing discipline process in the API is shown. The parameters of a *tbf* queuing discipline are in the format '*struct qdisc_tbf_usr*'. The format of the structure is given:

```
struct qdisc_tbf_usr
{
       __u32  limit;
       __u32  burst;
      __u32  burst_cell_log;
      __u32  rate;
       __u32  mtu;
      __u32  mtu_cell_log;
      __u32  mpu;
```

```
        __u32  peakrate;
        __u32  latency;
};
```

In order to create a *tbf* queuing discipline, the parameters of '*struct qdisc_tbf_usr*' are filled and the pointer to the element is copied in the *qdisc_options* field of '*struct qdisc_gen*'.

To perform operations on a class, '*struct class_msg*' element is used. For filters, '*struct filter_msg*' format is used in the API.

Once the parameters are filled, the values are passed into the kernel using system calls *set_qdisc*, *set_class* and *set_filter* for queuing discipline, class and filter respectively. The format is:

```
set_qdisc(cmd, (void *)&send_msg, sizeof(struct qdisc_gen));
```

The *cmd* can be one of ADD_QDISC, CHANGE_QDISC and DELETE_QDISC to create, modify and delete a queuing discipline. The term *send_msg* is an element of type '*struct qdisc_gen*'.

The formats for processing classes and filters are similar and can be found in the documentation.

The system call *set_qdisc* passes the control to *sys_set_qdisc( )* in net/qos_api/set_qdisc.c. The routine corresponding to the type of queuing discipline is called from *sys_set_qdisc( )* to process the arguments and call kernel QoS modules for further processing. To process *tbf* parameters, *fill_tbf_params( )* in net/qos_api/set_qdisc.c is used.

The design of the API for Linux QoS is provided in this section. These principles are used in making extensions to the API to accommodate delay, ber and serialq queuing disciplines.

This chapter gives an introduction to SBI emulation, details of SBI Node architecture and satellite link emulation. Also, Linux QoS modules for delay, ber and serialq queuing disciplines and interfaces for the modules are discussed.

3 Emulation of Ethernet Devices in SBI

The concept of Virtual Ethernet devices is introduced in the previous chapter. Ethernet devices are used to emulate satellite communication channels in the SBI emulation system. Instead of Physical Ethernet devices, Virtual Ethernet devices are used in the emulation. This chapter provides the details of the design of Virtual Ethernet Devices.

A Virtual Ethernet ('Veth') device is created over a Physical Ethernet ('Peth') device. Each Veth device has a unique Media Access Control (MAC) address. The main difference between a Veth and a Peth device is the absence of a physical driver in the former. A Veth device depends completely on the Peth device it is created on for data transfer. More than one Veth device can be created on a single Peth device. A sample configuration using Veth devices to increase the number of interfaces on a machine is shown in Figure 5. It shows various communication links possible with Veth devices created over Peth devices.

A preliminary design for Virtual Ethernet Devices is presented in [15]. Some of the structures used for Veth device emulation have a basis in the previous work.

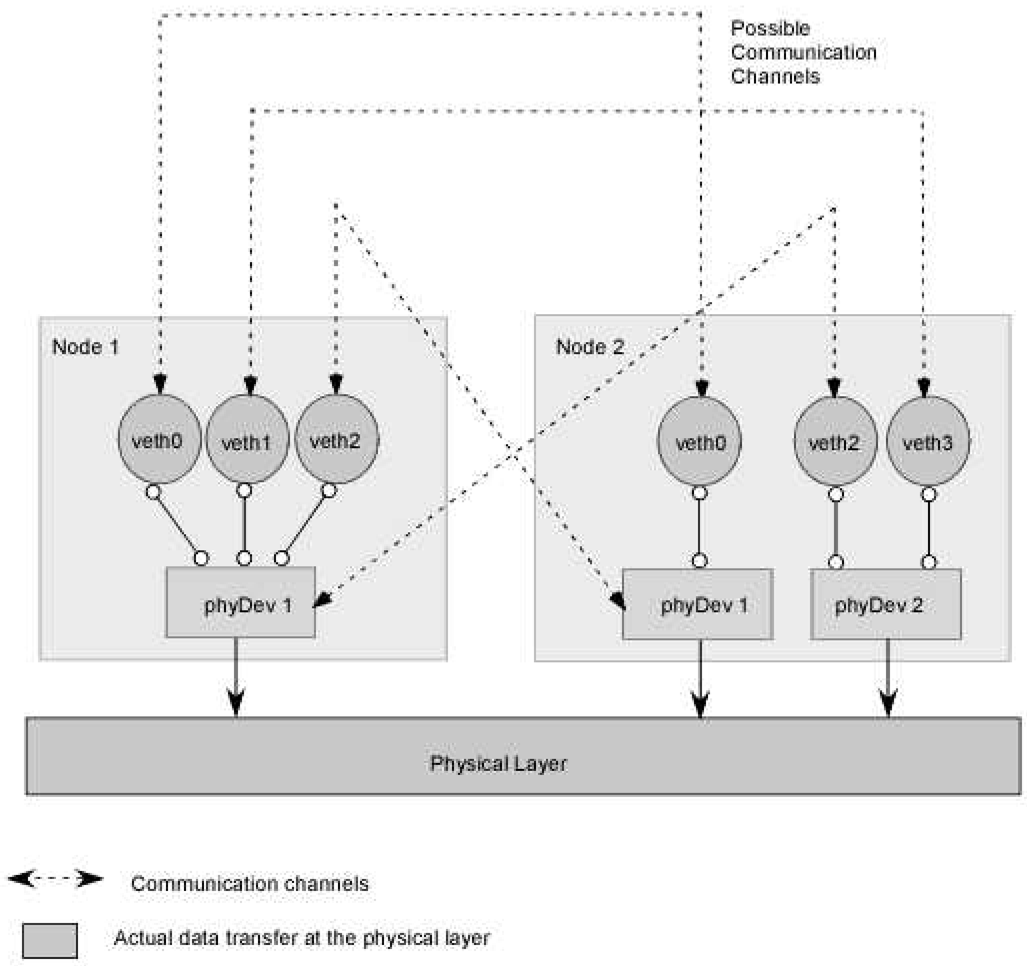


Figure 5: Sample Virtual Ethernet Configuration

3.1 Design Overview

Virtual Ethernet devices are emulated Ethernet devices. The processing of Veth devices is performed in the data link layer below the network layer in Linux kernel architecture as shown in Figure 6. The Veth device layer is transparent to the upper layers in the kernel architecture in the sense that the presence of the Veth layer does not affect the functioning of the upper layers.

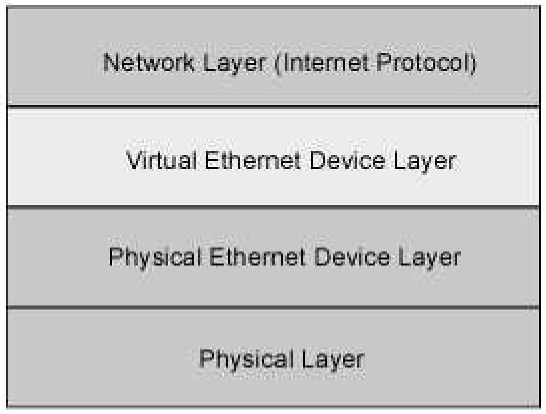

Network Layer (Internet Protocol)
Virtual Ethernet Device Layer
Physical Ethernet Device Layer
Physical Layer

Figure 6: Network Architecture with Virtual Ethernet Layer

A Veth device is created on an existing Peth device using ioctl calls from the user space. Once a Veth device is established as an Ethernet device, any packet transfer through it depends on the driver of the Peth device on which it is created. Though a Veth device emulates a Peth device, it is not a hardware device in the sense that it does not have a driver of it's own. Thus, a Peth device needs to be active for the Veth devices created on it to be accessible in a network. When a packet is sent from qdisc_restart to the Veth device layer, the packet is transmitted to underlying Peth device layer as shown in Figure 7.

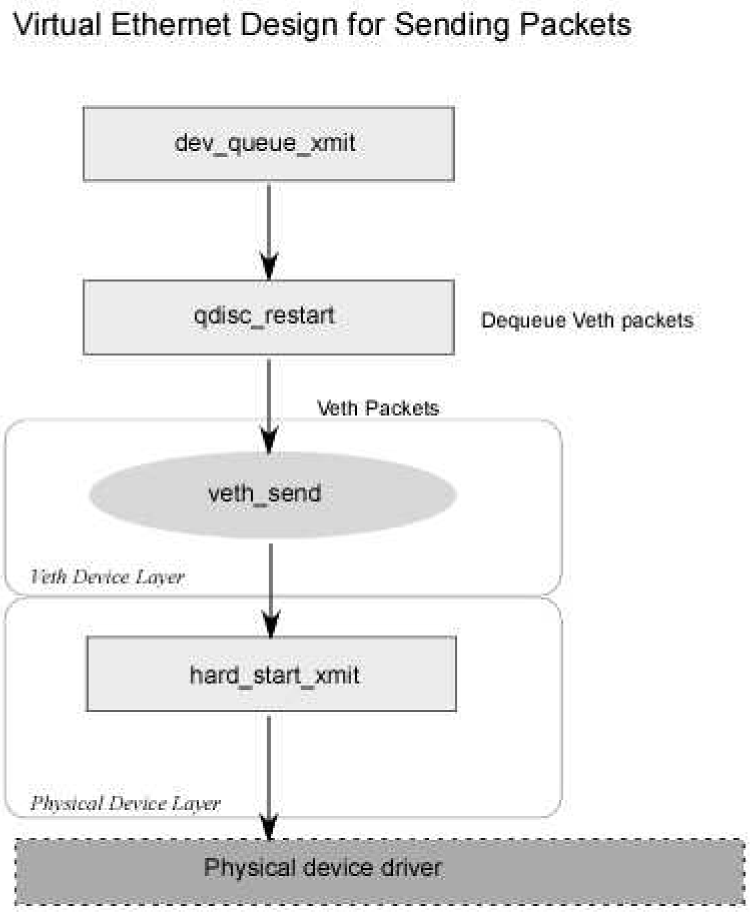


Figure 7: Packet Transmission Through Virtual Ethernet Device.

The architecture for receiving data for Veth devices is more elaborate because the data is obtained on the driver of a Peth device and the control over the operations in the physical layer is limited. The process of receiving Veth packets is shown in Figure 8. There are a couple of issues to be dealt with in the receiving of Veth packets.

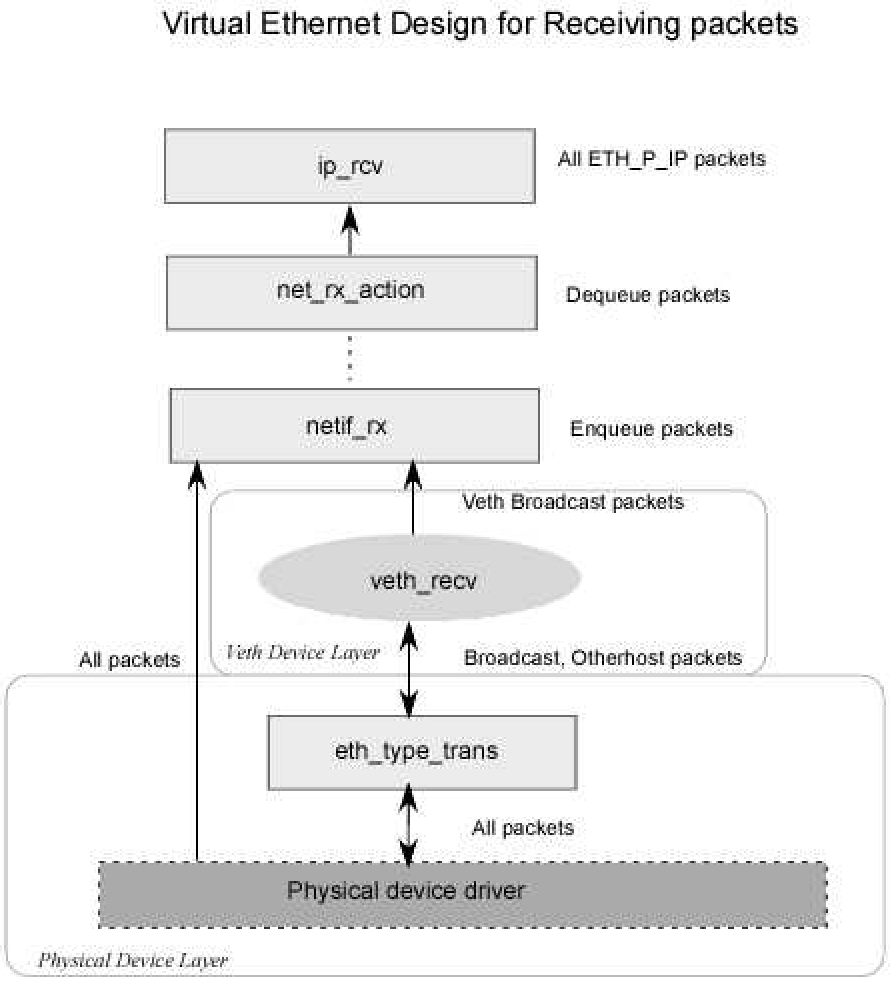


Figure 8: Receiving Virtual Ethernet Device Data

- Packets destined for Veth devices arrive on the driver of the underlying Peth device. Based on the MAC address of the destination on the packet, the packet type is determined. Any packet destined for Veth device has the MAC address of the Veth device and those packets are marked PACKET_OTHERHOST by Peth device. At the network layer, PACKET_OTHERHOST packets are directed back to the physical layer for forwarding to the network outside the node. Unless processing is done to save the Veth packets, they will be discarded from the node.

- Due to the absence of a driver, broadcast packets that arrive on the Peth device driver cannot reach the Veth devices unless specifically sent to the Veth devices created on the device.

These two issues are dealt with by forwarding PACKET_OTHERHOST and PACKET_BROADCAST packets from the Peth layer to the Veth layer. The implementation is given in the following section.

3.2 Virtual Device in Linux Kernel

The Veth devices are created, destroyed, modified and managed in the Linux kernel. The characteristics of an Ethernet device in Linux kernel are stored in the format '*struct net_device*' defined in include/linux/netdevice.h. Each ethernet device defines routines to open, close, set MAC address, receive data, send data and change MTU of the device.

Apart from the fields of '*struct net_device*', Veth devices have other elements to track like the underlying Peth device details and a linked list of all Veth devices *vethdev_list* similar to *dev_base* for Ethernet devices on a machine. The Veth characteristics are stored in the format '*struct veth_device*'.

```
struct veth_device
{
   char vethDevName[IFNAMSIZ];
   int itfNum;
   char phyDevName[IFNAMSIZ];
   struct net_device *vethDev;
   char srcMac[RAW_MAC_LENGTH];
   struct net_device_stats *vethStats;
   struct veth_device *next;
};
```

A short description for each term in the above structure is given below:

- The element *vethDevName* represents the name of the Veth device. The name of a Veth device is of the form '*veth#*' where # represents a number (the interface number). The first Veth device created on a node is named *veth0*, the next *veth1* and so on.
- *itfNum* is the interface number of a Veth device. For a Veth device named 'veth10', 10 is the *itfNum*. The first Veth device on a machine is assigned 0.
- *phyDevName* is the Peth device on which the present Veth device is created. An acceptable *phyDevName* is of the format 'eth#' where # is a number and '*eth#*' exists on the node.
- *vethDev* refers to the '*struct net_device*' element corresponding to the present Veth device. '*struct net_device*' element is generic to all the ethernet devices.
- *srcMac* is the MAC address of the Veth device in 6-byte format.

- *vethStats* is an element of format '*struct net_device_stats*' defined in include/linux/netdevice.h
- *next* points to the next element in the list of Veth devices *vethdev_list*. *vethdev_list* is a linked list of the Veth devices on a machine.

A list of Peth devices on which Veth devices are created is also maintained in the Veth device layer. The list is referred to as *phydev_list*. The format for storing the properties of Peth devices is:

```
Struct phy_device
{
   char phyDevName[IFNAMSIZ];
   struct net_device *phyDev;
   int num_vethdev;
   int (*change_mtu)(struct net_device*, int);
   struct phy_device *next;
};
```

The elements in '*struct phy_device*' are detailed:

- *phyDevName* is the Peth device corresponding to the '*struct phy_device*' element.
- *phyDev* is the '*struct net_device*' element corresponding to the Peth device.
- *num_vethdev* is the number of Veth devices created on this Peth device. Once the value of *num_vethdev* goes to 0, the *phyDev* elements are reset and the entry for the device is removed from *phydev_list*.

- *change_mtu* refers to the *change_mtu* pointer of *phyDev* prior to the creation of any Veth device on Peth device. This pointer is reset when *num_vethdev* reaches a value of 0.
- *next* points to the next element in the list of Peth devices *phydev_list*.

The parameters related to Veth devices passed between user and kernel space are in the formats provided below.

For the creation of a Veth device, '*struct veth_param_create*' defined in include/linux/veth.h is used.

```
struct veth_param_create
{
   int itfNum;
   char phyDevName[IFNAMSIZ];
   char srcMac[MAX_ADDR_LEN];
};
```

- The term *itfNum* is a number assigned to each Veth interface.
- Element *phyDevName* is the Ethernet device on which the Veth device creation is desired.
- Element *srcMac* is the MAC address of the Veth device to be created in 6-byte format.

For the deletion of a Veth device, '*struct veth_param_destroy*' defined in include/linux/veth.h is used.

```
struct veth_param_destroy
{
   int itfNum;
};
```

- The term *itfNum* is the interface number of the Veth device to be destroyed.

For listing Veth devices, '*struct veth_param_list*' defined in include/linux/veth.h is used.

```
struct veth_param_list
{
   int numDev;
   char list[LIST_SIZE];
};
```

- The term *numDev* represents the number of Veth devices on the node.
- Element *list* is a character array set to suitable size *LIST_SIZE* to fill the details of the Veth devices.

These structures are used through the implementation of the Veth devices.

3.2.1 Ioctl call handling

In the Linux kernel, ioctl calls made using sockets of protocol family PF_INET are handled by *inet_ioctl( )* routine in net/ipv4/af_inet.c. *inet_ioctl( )* handles *VETHDEV_CREATE*, *VETHDEV_DESTROY* and *VETHDEV_LIST* values of command *cmd* for Veth device layer once the values are defined. The values for the three commands are defined in include/linux/veth.h.

```
#define VETHIOC_ITF          0x9101
#define VETHDEV_CREATE      _IOW('q', VETHIOC_ITF, struct
veth_param_create)
+#define VETHDEV_DESTROY    _IOW('q', VETHIOC_ITF+1, struct
veth_param_destroy)
+#define VETHDEV_LIST       _IOW('q', VETHIOC_ITF+2, struct
veth_param_list)
```

The structure formats *veth_param_create*, *veth_param_destroy* and *veth_param_list* are used by applications to pass parameters through ioctl calls into the kernel to create, destroy and list Veth devices.

On receiving any of the three commands, *inet_ioctl( )* passes the control to *veth_ioctl(unsigned int cmd, void *arg)* defined in net/veth/veth.c.

The arguments in *arg* from user space are copied into kernel space for processing using *copy_from_user( )*. Depending on the command *cmd* used, *veth_ioctl( )* parses the arguments and calls appropriate methods for further processing.

3.2.1.1 ioctl create

The user space arguments are copied into kernel space using *copy_from_user( )*:

```
copy_from_user(param_create, arg, sizeof(struct
veth_param_create));
```

- *param_create* is an element of type '*struct veth_param_create*'
- *arg* is a pointer containing the arguments collected in the user space.

veth_ioctl( ) parses the *phyDevName* element of *param_create*. If the first three letters of *phyDevName* do not match with "*eth*", an error ENODEV is returned. If the *phyDevName* is valid, *veth_create( )* method is called for processing of the arguments and creation of Veth device.

3.2.1.2 ioctl destroy

The user space arguments are copied into kernel space using *copy_from_user( )*:

```
copy_from_user(param_destroy, arg, sizeof(struct
veth_param_destroy));
```

- *param_destroy* is an element of type '*struct veth_param_destroy*'
- *arg* is a pointer containing the arguments collected in the user space.

veth_ioctl( ) checks for the existence of a Veth device in *vethdev_list* that contains same *itfNum* element as *param_destroy->ITF_NUM*. If no Veth device exists with *itfNum*, an error ENODEV is returned. If a Veth device is found, *veth_destroy( )* method is called for processing of the arguments and deletion of the Veth device.

3.2.1.3 ioctl list

The user space arguments are copied into kernel space using:

```
copy_from_user(param_list, arg, sizeof(struct
veth_param_list));
```

- *param_list* is an element of type '*struct veth_param_list*'
- *arg* is a pointer containing the arguments collected in the user space.

veth_ioctl( ) calls *veth_list( )* method for processing the request to list Veth devices from user space.

3.2.2 Create

On obtaining control from *veth_ioctl( )*, creation of a Veth device is handled by *veth_create(struct veth_param_create *param_create)* defined in net/veth/veth.c.

A Veth device is created if valid Peth device and MAC address are given. On passing the control to *veth_create( )*, the following processing is done:

veth_create( ) checks if a Veth device exists in the *vethdev_list* with the *srcMac* passed from the user space.

- If a device is found, EEXIST error message is returned to indicate that the MAC address is not unique and the creation is thus stopped.
- If the *srcMac* requested for the creation of Veth device is unique, the process is continued.

An element *vethdevice* of format '*struct veth_device*' is created for the new Veth device to be created. The process is described below:

- Initialize the *init* pointer in *vethdevice->vethDev* element.

  ```
  vethdevice->vethDev->init = &veth_init;
  ```

- Set the Peth device on which the Veth device is going to be created in the element *phyDevName*.

- Set the interface number for the new Veth device. The first Veth device created on a node has an *itfNum* 0. For each of the remaining Veth devices, the *itfNum* is the maximum *itfNum* of the existing Veth devices incremented by 1.
- The name for the Veth device is the concatenation of "*veth*" and *itfNum* of the device.

  ```
  sprintf(vethdevice->vethDevName,"veth%d", vethdevice-
  >itfNum);
  sprintf(vethdevice->vethDev->name,"veth%d",vethdevice-
  >itfNum);
  ```

The remaining elements in *vethdevice* are filled during registration of the Veth device. The object *vethdevice* is then introduced at the head of *vethdev_list*. The routine *register_netdevice( )* defined in net/core/dev.c is called to register the Veth device.

If *register_netdevice( )* returns an error, the creation is stopped and the error is returned to *veth_ioctl( )*.

On success of the registration, the MAC address of the new Veth device is set. An object *hwaddr* of type '*struct sockaddr*' is required to set the MAC address. The element *hwaddr* is filled with Ethernet address family and the *srcMac*.

```
hwaddr->sa_family = ARPHRD_ETHER;
memcpy(hwaddr->sa_data, param_create->srcMac, vethdevice-
>vethDev->addr_len);
```

The MAC address is set to the device using the routine *veth_set_mac_address( )* described in the later sections.

```
veth_set_mac_address(vethdevice->vethDev, (void *)hwaddr);
```

The itfNum of the device is then set to *itfNum* element of *param_create* and a value of 0 is returned.

3.2.3 Initialize

The routine *register_netdevice( )* is called to register a Veth device in *veth_create( )*. r*egister_netdevice( )* uses the *init* pointer of '*struct net_device*' to initialize the characteristics of a device. The *init* pointer of a Veth device is referenced to *veth_init(struct net_device *vethDev)* in *veth_create( )*. *veth_init( )* sets the fields of the generic device structure '*struct net_device*'.

- The routines to open and close the device are set to *veth_open( )* and *veth_close( )* respectively.
- The *hard_start_xmit* is a function pointer used to transmit any data through the device. The routine *veth_send( )* is used for data transmissions through a Veth device.
- The routine *veth_get_stats( )* is used to obtain the statistics related to the Veth device.

  ```
  vethDev->get_stats = veth_get_stats;
  ```
- The fields of '*struct net_device*' that are generic to any device are set using *ether_setup( )* defined in drivers/net/net_init.c. Fields including *change_mtu*, *hard_header*, *set_mac_address*, *type*, *mtu*, *tx_queue_len* and *flags* are set in *ether_setup( )*.

  ```
  ether_setup(vethDev);
  ```

- *change_mtu* pointer in *vethDev* is set in *ether_setup( )* to *eth_change_mtu( )*. Since Veth devices require different processing for changing MTU, the pointer is changed and any manipulations with the MTU of the Veth device are handled by *veth_change_mtu( )*.

  ```
  vethDev->change_mtu = veth_change_mtu;
  ```

- Similarly, the *set_mac_address* is overwritten and the changes to MAC address of the Veth device are performed by *veth_set_mac_address( )*.

  ```
  vethDev->set_mac_address = veth_set_mac_address;
  ```

- The device flags are set to BROADCAST and MULTICAST in *ether_setup( )*. A Veth device is by default set only to BROADCAST by re-writing the field flags in vethDev.

- A queuing discipline is attached to each Ethernet device to queue outgoing packets set for transmission. For a Veth device created, no queuing discipline is set. Unless a queuing discipline is assigned to the Veth device, the default '*fifo*' qdisc is assigned when packets are transmitted.

  ```
  vethDev->qdisc_sleeping = &noop_qdisc;
  ```

Once the *vethDev* object is complete, the statistics field *vethStats* of '*struct veth_device*' is allocated memory and prepared for use.

The properties of the Peth device on which the Veth device is created are also defined in this section.

If a '*struct phy_device*' element *phydevice* already exists in *phydev_list* with the name *vethdevice->phyDevName*, the field *num_vethdev* of *phydevice* is incremented to indicate the number of Veth devices on the Peth device.

If no entry exists for the Peth device in *phydev_list*, an entry is created. The fields in '*struct phy_device*' are set as follows:

- The *phyDevName* is copied from *vethdevice->phyDevName*
- The '*struct net_device*' element *phyDev* is obtained using *dev_get_by_name( )* defined in net/core/dev.c

  ```
  phydevice->phyDev = dev_get_by_name(vethdevice-
  >phyDevName);
  ```
- The *num_vethdev* field is set to 1.

3.2.3.1 change_mtu

Any changes made to the MTU of a Peth device affect the Veth devices created on it. Hence, Peth device MTU handling should be made through *veth_change_mtu( )*. The original *change_mtu* pointer of the Peth device is stored in *change_mtu* field of *phydevice* for restoration after all the Veth devices on the Peth device are destroyed.

```
phydevice->change_mtu = phydevice->phyDev->change_mtu;
phydevice->phyDev->change_mtu = vethdevice->vethDev-
>change_mtu;
```

3.2.3.2 flags

The promiscuous flag on the Peth device is set in the initialization process. If the Peth device is not in promiscuous mode, packets for any device other than the Peth device are rejected at the physical layer. If the promiscuous mode is set, the packets for the Veth devices created over the Peth device are accepted. The *dev_change_flags( )* defined in net/core/dev.c handles changes made to flags in '*struct net_device*'.

```
dev_change_flags(phydevice->phyDev,  phydevice->phyDev->flags
| IFF_PROMISC);
```

Once the fields are filled, the *phydevice* is introduced into the *phydev_list*.

If the *phydev_list* exists, the *phydevice* is placed at the head of the list. If *phydev_list* does not exist, it is created.

The control is returned to *register_netdevice( )*. It appends the new Veth device at the tail of the device list *dev_base*. A return value of 0 is returned.

3.2.4 Destroy

veth_destroy(struct veth_param_destroy param_destroy)* is called by *veth_ioctl( )* in response to a request to destroy a Veth device. The element *param_destroy*, the pointer of type '*struct veth_param_destroy*' contains the interface number of the Veth device to be destroyed and is passed as the parameter. The process followed to destroy a Veth device in *veth_destroy( )*:

- The *vethdev_list* is scanned for an entry with the *itfNum* in *param_destroy*. The entry is removed from the list.

- The *phydev_list* is scanned for the Peth device on which the Veth device was created. The field *num_vethDev* in '*struct phy_device*' associated with it is decremented by 1. If the *num_vethDev* field reduces to 0,
  - The *change_mtu* pointer is restored in *phyDev* of the Peth device:

    ```
    phydevice->phyDev->change_mtu = phydevice->change_mtu;
    ```
  - The physical device is removed from *phydev_list*.

The Veth device is unregistered by calling *unregister_netdevice( )* in net/core/dev.c. On successful return from *unregister_netdevice( )*, the function call is returned with a value 0.

3.2.5 List

When the listing of Veth devices on a machine is requested, *veth_list(struct veth_param_list* param_list)* is called by *veth_ioctl( )*.

- The *list* field in *param_list* is filled with the details of each of the Peth devices on which Veth devices are created and the number of Veth devices on each of them. The interface number, name and MAC address of each Veth device is appended to the *list* field.
- The *numDev* field in *param_list* is set to the number of existing Veth devices.

The function call is then returned with a value 0.

3.2.6 Open

When a Veth device is brought up by the *ifconfig* command in the user space, the function *veth_open(struct net_device* device)* is invoked. A sample command:

```
ifconfig veth3 up
```

This routine enables the queue associated with the Veth device pointed to by the '*struct net_device*' pointer for data transfer.

```
netif_start_queue(device);
```

3.2.7 Close

A request from the user space using *ifconfig* command to bring down a Veth device, i.e., close the network access of the Veth, the *veth_close(struct net_device *device)* function is called. This routine stops the queue associated with the Veth device pointed to by the '*struct net_device*' pointer.

```
netif_stop_queue(device);
```

3.2.8 Change MTU

Any changes in MTU of a Veth device and the underlying Peth device are handled by *veth_change_mtu(struct net_device *device, int new_mtu)*. The generic function for managing changes in MTU is *eth_change_mtu( )*. Since Veth devices are created over Peth devices, the MTU of the Veth devices should be less than or equal to the MTU of the Peth device on which it is created. To ensure this, the changes to MTUs of both Veth and Peth devices are handled by *veth_change_mtu( )*.

Since the *change_mtu* function pointers of both the Veth device and the underlying Peth device point to *veth_change_mtu( )*, the request for MTU change can be from a Veth or a Peth device. Whether the device is a Veth or a Peth device can be found by searching the *vethdev_list* or *phydev_list* for existence of the device.

If the device that requested change of MTU is a Veth device,

- If the *new_mtu* is greater than the MTU of the Peth device on which it is created or lesser than 68 bytes, an error EINVAL is returned.
- If the *new_mtu* meets the requirement specifications compared with the MTU of the Peth device, the *new_mtu* is set to the Veth device.

If the device that requested change of MTU is a Peth device,

- The MTU of all the Veth devices created on it must be lesser than or equal to the MTU of the physical device.
- If the MTU of a Veth device is greater than the *new_mtu* of the Peth device, the MTU of the Veth device is set to *new_mtu*.
- The MTU of the Peth device is set to *new_mtu*.

On successful assignment of the MTU to the device, 0 is returned to the calling function.

3.2.9 Change MAC address

On the creation of a Veth device, *veth_create( )* sets the MAC address using the routine *veth_set_mac_address(struct net_device *vethDev, void *hw_addr)*. The MAC address is given in the form of *hw_addr*. The MAC address is set for the Veth device:

```
memcpy ( vethDev->dev_addr, mac_addr->sa_data, vethDev-
>addr_len );
```

The 6-byte MAC address *vethDev->dev_addr* is then converted into readable format HH:HH:HH:HH:HH:HH and stored in *srcMac* field of '*struct veth_device*' element associated with the Veth device.

3.2.10 Obtain the Statistics

The routine *veth_get_stats(struct net_device* device)* is called when an *ifconfig* command is executed over a Veth device. This function returns the pointer to *vethStats* element of type '*struct net_device_stats*' that contains the statistics pertaining to the Veth device. The pointer, *vethStats* is an element in '*struct veth_device*' associated with the Veth device.

3.2.11 Transmit data

When data is set for transmission on a device, the *hard_start_xmit* function associated with the device is called. In case of Veth devices, *veth_send(struct sk_buff *skb, struct net_device *device)* is called by *qdisc_restart( )* defined in net/sched/sch_generic.c. The parameters that are passed are the packet *skb* and the '*struct net_device*' element of the Veth device on which the packet is to be sent.

The design for transmitting data is provided below:

- The '*struct net_device*' structure of the Peth device on which the Veth device is created is obtained using *dev_get_by_name( )* in net/core/dev.c.
- If the Peth device is up, the packet structure *skb* is passed onto *hard_start_xmit( )* of the Peth device. The packet is sent over the device driver of the Peth device and the *vethStats->tx_packets* field of '*struct veth_device*' is incremented. The

return value obtained from *hard_start_xmit* of the physical device is returned to the calling function.

- In case the Peth device is busy, a return value of –1 is returned.

3.2.12 Receive data

As described in the previous sections, the design for receiving data for Veth devices takes into consideration the fact that processing in the physical Ethernet device layer needs to be performed. The process of receiving Veth packets is shown in Figure 8.

When the physical device receives a packet, it calls *eth_type_trans( )* in net/ethernet/eth.c to set the protocol ID and packet type – host, broadcast or other host packet. Once the packet type is set, the packet is sent back to the physical device receive modules by *eth_type_trans( )*. The modules then put the packet in a queue by calling *netif_rx( )* in net/core/dev.c as shown in Figure 8.

It is from *eth_type_trans( )* that the Veth device layer receive modules are contacted. Modifications are made to *eth_type_trans( )* change the packet type of Veth packets to PACKET_HOST and to receive broadcast data for Veth devices.

3.2.12.1 eth_type_trans

The function *eth_type_trans(struct sk_buff *skb, struct net_device *device)* is implemented in net/ethernet/eth.c. The routine is used to set a packet's protocol ID and packet type. Modifications have been made to the routine to accommodate proper handling of packets for Veth devices.

Since the packets for Veth devices have the MAC address of a Veth device, the packets *skb->pkt_type* is set to PACKET_OTHERHOST and the broadcast packets have *skb->pkt_type* set to PACKET_BROADCAST. The implementation is extended and the packets that are labeled PACKET_BROADCAST or PACKET_OTHERHOST are sent to *veth_recv( )* in net/veth/veth.c to filter out the packets for Veth devices.

3.2.12.2 veth_recv

veth_recv(unsigned char dev_addr, unsigned short protocol, struct sk_buff *skb)* is called by *eth_type_trans*( ) in net/ethernet/eth.c. The parameters that are passed are the MAC address on the packet, *dev_addr*, the packet protocol and the packet structure *skb*.

- If the *skb->pkt_type* is PACKET_OTHERHOST, the MAC address *dev_addr* on the packet is compared with the MAC address of each Veth device in *vethdev_list*. If the packet is sent for one of the Veth devices, the packet will contain a Veth device MAC address. Once the device is found, the *skb->dev* field in the packet is set to the Veth device pointer *vethDev* field in '*struct veth_device*' and the *skb->pkt_type* is changed from PACKET_OTHERHOST to PACKET_HOST. The packet is now indicates that it's been received for a device on the node and the new packet type allows further processing at the network layer. The function call is returned with a value 0. On returning to the Peth device modules, the packet is inserted into a receive queue by calling *netif_rx( )* in net/core/dev.c.

- If the *skb->pkt_type* is PACKET_BROADCAST, a copy of the packet is made for each of the Veth devices over the Peth device in context. In each of the copies of the packet, the *skb->dev* field is set to the *vethDev* field of the '*struct veth_device*' element associated with the Veth device. The copy is placed in the generic receive queue by calling *netif_rx( )* in net/core/dev.c. The function call is returned with the return value of *netif_rx( )*.

If the packet is intended for any of the Veth devices, the statistics are updated, i.e., the field *vethStats->rx_packets* in the '*struct veth_device*' structure is modified. Any processing of the received packet can be performed in this module.

3.2.13 Modifications to Linux kernel

The implementation of Virtual Ethernet devices is made in Linux kernel version 2.4.6. The parts of the kernel modified are listed:

- The routine *inet_ioctl( )* in net/ipv4/af_inet.c is modified to process ioctl calls for Veth devices.
- The routine *eth_type_trans( )* in net/ethernet/eth.c is modified to direct Veth and broadcast packets to Veth device layer.

3.3 User Interface for Veth Devices 'Vethctl'

Veth Devices are created using ioctl calls in the user space. The user interface provides a command '*Vethctl*' that can be used to create, destroy and list Veth devices on a machine.

For the ioctl operations, a socket is created in protocol family PF_INET.

```
ioctl_fd = socket(PF_INET, SOCK_DGRAM, 0);
```

The ioctl call from the user control program is received by the *inet_ioctl( )* function in net/ipv4/af_inet.c. For the following ioctl commands, Veth modules are called for further processing by *inet_ioctl( )*.

VETHDEV_CREATE	-	to create a Veth device
VETHDEV_DESTROY	-	to destroy a Veth device
VETHDEV_LIST	-	to list all the Veth devices

3.3.1 Creation

The format for creating a Veth device using Vethctl is:

```
Vethctl -c   PDEV_NAME    MAC_ADDRESS
```

- 'c' option is used to create a Veth device
- *PDEV_NAME* is the name of an existing Peth device on which a Veth device is to be created
- *MAC_ADDRESS* is the MAC address in the format HH:HH:HH:HH:HH:HH. The MAC address to be assigned should be unique.

Vethctl application parses the arguments given using *Vethctl* and fills '*struct veth_param_create*' element *param_create*.

```
strcpy (param_create.phyDevName ,argv[2]);
convMac(argv[3], param_create.srcMac);
```

convMac( ) is a routine used to convert the MAC address given in HH:HH:HH:HH:HH:HH format to 6-byte hardware address format.

The element *param_create* is then passed into the kernel using ioctl call. The ioctl call made to the kernel to create Veth device is

```
return_value = ioctl(ioctl_fd, VETHDEV_CREATE, (char
*)&param_create);
```

- If the ioctl call is successful, 0 is returned from the kernel and the term *itfNum* is filled in the kernel. The name of the Veth device created is the concatenation of '*veth*' and *itfNum*. If the *itfNum* is 3, the Veth device created is *veth3*. A typical output would be:

  ```
  vethctl: Virtual Device veth3 created successfully
  ```

- In case of error, the ioctl may return a variety of error values. The interpretation of the messages related to *Vethctl* is listed:

  ENODEV - No physical device found with name PDEV_NAME in the list of physical ethernet devices.

  EEXIST - A Veth device already exists with MAC_ADDRESS.

3.3.2 Destruction

The format for destroying a Veth device using Vethctl is:

```
Vethctl -d   ITF_NUM
```

- 'd' option is used to destroy an existing Veth device
- *ITF_NUM* is the interface number of an existing Veth device. If the Veth device to be destroyed were *veth4*, *ITF_NUM* would be 4. A sample command:

  ```
  # vethctl -d 4
  ```

Vethctl application parses the arguments given using '*Vethctl*'.

- If the *ITF_NUM* is not a digit, Vethctl returns an error.
- If the *ITF_NUM* is valid, Vethctl fills '*struct veth_param_destroy*' element *param_destroy*.

The field of *param_destroy* is filled.

```
param_destroy.itfNum = atoi(argv[2]);
```

The element *param_destroy* is then passed into the kernel using ioctl call. The ioctl call made to the kernel to destroy a Veth device is

```
return_value = ioctl(ioctl_fd, VETHDEV_DESTROY, (char
*)&param_destroy);
```

- On successful deletion of the Veth device, a value 0 is obtained. The output would be:

  ```
  vethctl: Virtual Device veth4 destroyed successfully
  ```

- In case of error, the ioctl may return ENODEV, the interpretation is given:

  ENODEV - No device with name *veth<ITF_NUM>* found in the list of Veth devices.

  The other error messages are not directly related to the parameters passed using Vethctl.

3.3.3 List

The format for requesting a list of Veth devices using Vethctl is:

```
Vethctl -l
```

- 'l' option is used to list existing Veth devices.

Vethctl passes an empty element of format '*struct veth_param_list*' *param_list* to the kernel space using ioctl call. The fields of *param_list* are filled in the kernel.

The ioctl call made to the kernel to list all Veth devices on a node is

```
return_value = ioctl(ioctl_fd, VETHDEV_LIST, (char
*)&param_list);
```

On success, a value 0 is obtained. The *numDev* field is set to number of Veth devices on the node and the element list is filled with the details of Veth devices in the kernel.

A typical output would be:

```
Number of veth devices on this host: 4

List of devices:

On eth1 :        4 virtual devices

  Virtual device  Physical device  itfNum    Mac address

     veth3           eth1            3        00:04:86:00:15:03

     veth2           eth1            2        00:04:86:00:15:02

     veth1           eth1            1        00:04:86:00:15:01

     veth0           eth1            0        00:04:86:00:15:00
```

In case of error, messages that are not directly related to the parameters passed using *Vethctl* may be returned.

3.3.4 Configure

The command *ifconfig* in the user space provides a variety of generic operations to configure an existing Ethernet device. Some of the operations permitted using *ifconfig*:

- Network status change - bring a device up or down
- Set IP and broadcast addresses
- Modify MAC address
- Change Maximum Transfer Unit (MTU)
- Set promiscuous mode.

The operations can be performed on the created Veth devices to configure the device properties.

3.4 Conclusion

This chapter gives a detailed design and implementation of Virtual Ethernet devices that emulate Physical Ethernet devices. The implementation is tested and the results are included in Chapter 8.

4 Emulation of Propagation Delay in SBI

The data transmitted between satellites in space is subject to transmission and propagation delays due to the communication link between the satellites. The transmission delay results due to transmission through a link. The propagation delay results due to the distance travelled by the packet from the source to the destination.

In the real world SBI system, data transmission is between two satellites or a satellite and a ground station. Transmission delay results due to the transmission of a packet onto the physical Ethernet medium of transfer. This is equal to

$$\text{Transmission Delay} = \frac{\text{Packet Length}}{\text{Link Speed}}$$

The propagation delay is calculated using the distance between communication elements and the rate of data transfer.

$$\text{Propagation Delay} = \frac{\text{Distance between Satellites in Space}}{\text{Data Transfer Rate}}$$

The propagation delay in packet flow from a ground station to a GEO satellite is about 130ms. It is relatively high compared to typical terrestrial network propagation delay.

In the SBI emulation system, each Ethernet device emulates a communication channel on a satellite with appropriate link rates and bandwidth. Hence, the two constituents of delay in satellite environment need to be introduced into the SBI emulation system.

- The transmission delay is emulated in the system by using device link rates comparable to space communication link rates.
- The propagation distances between two nodes in SBI emulation system are considerably lower than inter-satellite distances. Hence, propagation delay cannot be emulated into the system using distance as a factor. Instead, it is introduced into the emulation system before the packet flow is transmitted on an interface.

4.1 Design Overview

On each Ethernet device used as a communication link in SBI emulation, packets are queued onto an outgoing queue before transmission. The simulation of propagation delay is implemented by delaying packets in the outgoing queue by the amount of propagation delay calculated for the particular interface acting as communication link in the emulation system. The implementation is described in the next section.

The value of propagation delay inserted into packet flow can range from 1 msec to 300msec (tested values). The resolution of the delay is dependent on the system jiffy. A typical value of jiffy is 10msec in Linux 2.4.6. The value can be changed to 1msec to increase the resolution of propagation delay.

4.2 Propagation Delay in Linux Kernel QoS

In the Linux kernel QoS, each Ethernet device queues packets for transmission using one of the queuing discipline (qdisc). The default queuing discipline is 'fifo' – First

In First Out. Each qdisc applies it's own properties on the packet queue. To insert propagation delay into a packet flow, '*delay*' qdisc is designed.

The principle based on Linux QoS details given in previous chapters is described:

When a packet is enqueued into *delay* qdisc, the entry time is recorded in jiffies. The exit time in jiffies of the packet is calculated by adding the delay value to the entry time. The packet is dequeued at the exit time.

When a qdisc is created in Linux Kernel QoS, elements of type '*struct Qdisc*' and '*struct Qdisc_ops*' are attached to it. For the '*delay*' qdisc, the relevant routines in '*struct Qdisc_ops*' used in the functioning of the qdisc are: *delay_init*, *delay_enqueue*, *delay_dequeue*, *delay_requeue*, *delay_change* and *delay_destroy*.

A structure of the format '*struct delay_sched_data*' that contains the attributes of delay qdisc is created:

```
struct delay_sched_data
{
   unsigned long delay_jiffs;
   struct delay_track   *list_head;
   struct delay_track   *list_tail;
   struct timer_list    sleep_timer;
};
```

- The *delay_jiffs* field contains the relative delay value to be implemented on the packets transmitted by the device *dev* i.e., the packets are delayed by an amount delay from the time of entry into the device queue. The value is defined in jiffies.
- The *list_head* field points to the head of the Packet Entry Queue (PEQ). It is a list of objects of '*struct delay_track*' type.

- The *list_tail* field points to the tail of the PEQ as shown in Figure 9.
- The *sleep_timer* field contains parameters associated with the sleep of the dequeue process till the packet at the head of the list is to be dequeued.

Elements of type '*struct delay_track*' form the PEQ. Each element of PEQ is unique for a set of consecutive packets that have the same exit time. The details of '*struct delay_track*':

```
struct delay_track
{
      struct delay_track   *prev;
      struct delay_track   *next;
      unsigned long  exit_jiffs;
      int num_pkts;
};
```

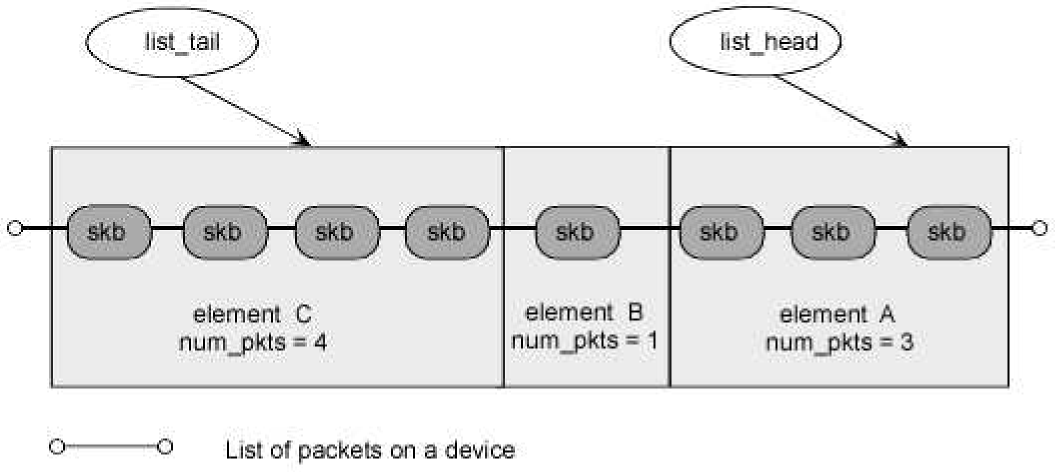


Figure 9: Sample Arrangement of Packet Entry Queue

- The *prev* field points to the previous entry in the PEQ.

- The *next* field points to the next entry in the PEQ.
- The *exit_jiffs* field is the absolute time at which the packet(s) should be dequeued from the packet queue.
- The *num_pkts* field is the number of packets to which, the fields in the entry apply.

In the sample PEQ given in Figure 9, the *prev* field in entry B would be A. The *next* field in entry B would be C.

The parameters related to delay qdisc are passed between user and kernel spaces in the format '*struct tc_delay_qopt*':

```
struct tc_delay_qopt
{
   int delay_msec;
};
```

- The term *delay_msec* is the relative value of delay in milliseconds.

4.2.1 Initialize

When a delay qdisc is created on an Ethernet device, the fields of '*struct delay_sched_data*' are initialized in *delay_init(struct Qdisc *sch, struct rtattr *opt)* defined in net/sched/sch_delay.c. An element *q* of type '*struct delay_sched_data*' is filled during initialization. The *data* pointer in '*struct Qdisc*' element corresponding to *delay* qdisc points to *q*. The term *opt* contains the parameters passed from the user space.

The parameters are obtained in '*struct tc_delay_qopt*' element *ctl* from the element *opt*. The user control program passes the value of delay in milliseconds. The delay value is converted to jiffies and stored in *delay_jiffs* element of *q*.

The *list_head* and *list_tail* point to the head and tail of PEQ and are set to NULL during initialization. The *sleep_timer* is initialized to

```
q->sleep_timer.data = (unsigned long)sch;
q->sleep_timer.function = delay_watchdog;
```

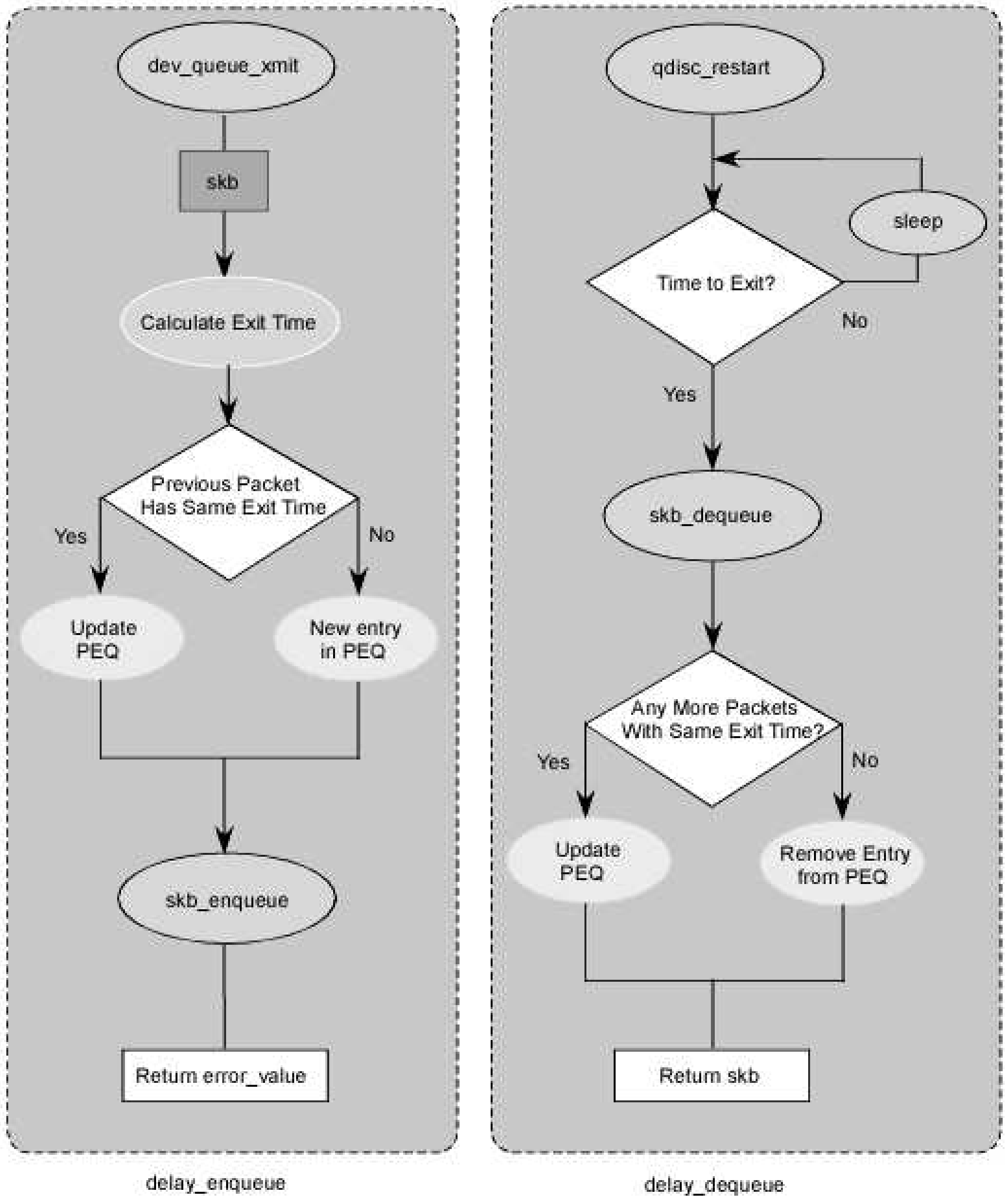
Note: *delay_watchdog( )* routine calls the *netif_schedule( )* function for the device on which the delay qdisc is created. *Netif_schedule( )* calls *net_tx_action( )* through soft interrupts that inturn calls *qdisc_run( )* and *qdisc_restart( )*. This schedules the call to the dequeue function of delay qdisc. The routines *delay_enqueue* and *delay_dequeue* functions are shown in Figure 10.

4.2.2 Enqueue

When a packet is set for transmission by a device on which delay qdisc has been attached, the *delay_enqueue(struct sk_buff *skb, struct Qdisc *sch)* routine is called by *dev_queue_xmit* to enqueue the packet into delay queue. The term *skb* represents the packet to be transmitted.

The *struct delay_sched_data* object *q* associated with the device on which the delay qdisc is created is obtained from *sch* as:

```
q = (struct delay_sched_data *)(sch -> data);
```



PEQ - Packet Entry Queue

Figure 10: Delay Queuing Discipline Enqueue and Dequeue

The *exit_jiffs* at which the packet has to be transmitted (dequeued from the delay queue) is found by:

```
exit_jiffs = jiffies + q->delay_jiffs
```

The element *jiffies* gives the current system time in jiffies. Jiffies and absolute value of delay for the device are used to calculate the *exit_jiffs* for the packet.

- If the PEQ is empty (*q->list_tail* is NULL) or if the PEQ is not empty but the *exit_jiffs* of the last element in PEQ (*q->list_tail*) does not match with the *exit_jiffs* of the packet, a new entry *new_entry* of '*struct delay_track*' is made. The fields of *new_entry* are filled as follows:

  ```
  new_entry -> num_pkts = 1;
  new_entry -> exit_jiffs = exit_jiffs
  ```

  The new element is attached to the tail of PEQ.

- If the PEQ is not empty (*q -> list_tail* is not NULL) and the *exit_jiffs* of *q -> list_tail* matches with the *exit_jiffs* of the packet, the *q -> list_tail* is updated, the number of packets *num_pkts* in the last entry is incremented by 1.

Now that the entry for the exit time of a packet is made, the packet is enqueued onto the packet queue defined by *sch->q* using *skb_queue_tail* in include/linux/skbuff.h.

```
skb_queue_tail(&sch->q, skb);
```

The backlog, bytes and packets fields of *qdisc_delay* are updated.

On successful enqueue of the packet, a value of 0 is returned to the calling function.

4.2.3 Dequeue

The function *delay_dequeue(struct Qdisc *sch)* is invoked by *qdisc_restart( )* in net/sched/sch_generic.c to dequeue a packet from delay queue for transmission. The '*struct delay_sched_data*' element *q* associated with the device on which the delay qdisc is created is obtained from *sch* as follows:

```
q = (struct delay_sched_data *)sch->data;
```

If the PEQ is not empty (*q->list_head* is not NULL), the *q->list_head->exit_jiffs* is compared with the present time *jiffies*.

- If the *q->list_head->exit_jiffs* is greater than the present jiffies, no packet from delay queue is dequeued. Hence, the dequeue process is made to sleep until *q->list_head->exit_jiffs*.

  ```
  mod_timer(&q->sleep_timer, q->list_head->exit_jiffs);
  ```

- If the *q->list_head->exit_jiffs* is lesser than or equal to the present jiffies, one packet in the entry at the head of PEQ is dequeued using *skb_dequeue( )* implemented in include/linux/skbuff.h. The number of packets in this entry is decremented by 1.

  ```
  skb = skb_dequeue(qdisc_delay->q);
  q -> list_head -> num_pkts--;
  ```

If the num_pkts of *q->list_head* reduces to 0, the entry at the head of PEQ is removed and *q->list_head* is updated.

The timer is set to the *q->list_head->exit_jiffs* to schedule a call to the dequeue function for the next packet in the queue.

```
mod_timer(&q->sleep_timer, q->list_head->exit_jiffs);
```

Hence, the *delay_dequeue( )* can be called either due to the interrupt created by the timer or by the repeated calls to *qdisc_restart( )* by *qdisc_run( )*. Either way, the *exit_jiffs* is compared with *jiffies* and packet is dequeued based on the appropriate delay value set to it on the entry to delay queue.

4.2.4 Requeue

Once a packet is dequeued from delay queue, the *hard_start_xmit* of the device on which the delay qdisc is created is called. If the device is busy, the packet is requeued for later transmission. The *delay_requeue(struct sk_buff *skb, struct Qdisc *sch)* method is called to requeue a packet onto the delay queue.

```
skb_queue_head(&sch->q, skb);
```

4.2.5 Destroy

A qdisc is destroyed upon request from user space using '*tc*' command or when a system is set to reboot. This action is handled by *delay_destroy(struct Qdisc *sch)*. The *sleep_timer* in '*struct delay_sched_data*' is destroyed and the module count is decremented.

```
del_timer(&q->sleep_timer);
```

4.2.6 Modify attributes

A modification to the delay qdisc attributes is a change in the value of propagation delay implemented on the packet flow transmitted by a device.

Upon request for a change from the user space, *delay_change(struct Qdisc *sch, struct rtattr *opt)* is called to change the value of *delay_msec* field of '*struct delay_sched_data*'.

On obtaining the delay value from user space, it is converted into jiffies and stored in delay_jiffs.

```
q->delay_jiffs = timespec_to_jiffies(&delay_value_timespec);
```

4.2.7 Display attributes

A request from user space to list the attributes of delay qdisc results in a call to *delay_dump(struct Qdisc *sch, struct sk_buff *skb)* routine. The attribute of delay qdisc, i.e., the value of propagation delay set to the packet flow on a device is returned in terms of milliseconds.

```
opt.delay_msec = (delay_value_timespec.tv_sec * 1000) +
(delay_value_timespec.tv_nsec / 1000000);
```

4.2.8 Modifications to Linux kernel

The implementation of propagation delay queuing discipline is made in Linux kernel version 2.4.6. The parts of the kernel modified are listed:

- The definition of structure 'tc_delay_qopt' is made in include/linux/pkt_sched.h.

4.3 Iproute Interface

An interface for *delay* qdisc is created using Iproute. Delay qdisc can be created, modified, destroyed and it's attributes displayed using Iproute interface.

The format for setting the delay value of *delay* qdisc using *tc* command is:

```
tc qdisc add <DEV_HANDLE> delay delay_msec DELAY_VALUE_IN_MSEC
```

The term DEV_HANDLE is defined in Chapter 2.

The format for changing the propagation delay value of *delay* qdisc using *tc* command is:

```
tc qdisc change <DEV_HANDLE> delay delay_msec
DELAY_VALUE_IN_MSEC
```

The format for destroying the *delay* qdisc using *tc* command is:

```
tc qdisc del <DEV_HANDLE> delay
```

The format for displaying the attributes of all qdiscs on the node using *tc* command is:

```
tc qdisc show
```

The parameters given using '*tc*' command are parsed using the routine *delay_parse_opt( )* created in tc/q_delay.c. The *delay_msec* field in '*struct tc_delay_qopt*' element is filled and passed into the kernel using netlink calls.

```
qopt.delay_msec = (atoi)(*argv);
addattr_l(n, 1024, TCA_OPTIONS, &qopt, sizeof(qopt));
```

4.4 API for QoS interface

The API for QoS is used to dynamically set the QoS attributes of interfaces. A background on the API is given in the Chapter 2. The API is extended to create, modify and delete *delay* qdisc through an interface.

The parameters required to process *delay* qdisc operations are in the format '*struct qdisc_delay*' similar to '*struct tc_delay_qopt*' for compatibility.

```
struct qdisc_delay
{
      int delay_msec;
};
```

- *delay_msec* is the value of propagation delay to be set on the interface desired.

To create, delete or modify a *delay* qdisc, an element *send_msg* of format '*struct qdisc_gen*' is filled with appropriate values. A pointer to *delay_opt* of type '*struct qdisc_delay*' is linked to it.

```
delay_opt.delay_msec = delay;
send_msg.qdisc_options = (void *)&delay_opt;
```

Using system call *set_qdisc*, the parameters are passed into kernel space. The operations are indicated using commands ADD_QDISC, DELETE_QDISC and CHANGE_QDISC. As a sample, creation operation of *delay* qdisc is shown.

```
set_qdisc(ADD_QDISC, (void *)&send_msg, sizeof(struct
qdisc_gen));
```

In net/qos_api/set_qdisc.c, a routine *fill_delay_params( )* is created to process API requests for *delay* qdisc. The user space arguments are processed and *usropt* of '*struct qdisc_delay*' format is filled.

Using rtattr objects, internal kernel QoS modules are called for further processing of the queuing discipline.

```
fill_rtattr(&fill_param->nlm, 1024, TCA_OPTIONS, usropt,
sizeof(struct qdisc_delay));
```

The *delay* qdisc can thus be created, deleted or modified using the programming interface. This interface is used in the SBI emulation scenario to operate *delay* qdiscs on interfaces.

4.5 Conclusion

This chapter gives a detailed design for the introduction of propagation delay into a packet flow. The Linux QoS modules are extended in the creation of delay queuing discipline. The implementation is tested and the results are included in Chapter 8.

5 Emulation of Bit Error Rate in SBI

Data flow between a transmitter and receiver is subject to loss due to various factors like attenuation, atmospheric absorptions and medium of propagation. The degree of loss is dependent on factors like the medium of transfer and the distance between the elements of data transfer.

Bit Error Rate (BER) is used to represent the degree of loss of data. BER is the ratio of error bits received to the total number of bits received. If 2 bits are in error in 10,000,000 bits received, the BER is 2*10e-7.

In case of communication between satellites, data is affected by the medium of access and the large distances for the data to propagate. Due to the great distances between elements of data transfer in space, the BER is considerably higher than that results in transfer between two nodes used in SBI emulation. Hence, introduction of BER into packet flow is essential to emulate BER into SBI system.

5.1 Design Overview

The Bit Error Rate is emulated in the SBI system by introducing error in randomly selected bits in the packet flow. On each Ethernet device used as a communication link in SBI emulation, a queuing discipline '*ber*' is attached. The packets transmitted by each Ethernet device are queued onto the *ber* qdisc. In the *ber* qdisc, the number of bits erred in a packet flow is proportional to the value of BER associated with the communication link. The implementation is described in the next section.

5.2 Bit Error Rate in Linux Kernel QoS

As explained in the previous chapter, an Ethernet device queues packets for transmission using a queuing discipline (qdisc) in Linux kernel QoS. Ber qdisc is created to introduce bit errors into data flows through the device on which it is created. The ber qdisc is attached to a device involved in SBI emulation and the packet flow is erred depending on the ber value set for communication through the device.

When a qdisc is created in Linux Kernel QoS, elements of formats '*struct Qdisc*' and '*struct Qdisc_ops*' are attached. For the 'ber' qdisc, the routines used to perform operations on the qdisc are: *ber_init*, *ber_enqueue*, *ber_dequeue*, *ber_requeue*, *ber_change* and *ber_destroy*.

When a packet is set for transmission on a device on which ber qdisc has been attached, the packet is placed in ber queue. A random number generator is used to obtain a random number between 0 and the BER value for the device. If the random number lies in the packet data, an error is introduced into the packet. The bit corresponding to the random number is flipped (1 becomes 0 and vice versa) and the packet is transmitted.

On the receiver side, the packet is rejected due to the checksum failure. Since one bit error causes a failure in the checksum, a maximum of one bit error is introduced into a packet. As the bit error for packet flow increases, the chance of a packet getting erred increases. Also, as the length of a packet increases, there's a higher chance of getting erred. Thus depending on the intensity of bit errors desired, bits are erred and lead to loss of data.

In a reliable protocol like TCP, a retransmission of the packet is required for each packet containing error. During retransmission, the error inserted into a packet is reset, the process is reversed bringing the packet back to clean state.

The details of ‘*struct ber_sched_data*’ that contains the attributes of ber qdisc:

```
struct ber_sched_data
{
      int ber_value;    /* ber value given by the user */
};
```

- The *ber_value* field is the bit error value passed from the user space. The packet flow will be introduced with error in 1 bit out of *ber_value* bits in data transfer.

Whenever a bit is erred in reliable transmission (TCP), the details of the packets that are erred are recorded. Each packet details are stored in the format ‘*struct error_skb*’ and stored in a list *err_list*. The details of ‘*struct error_skb*’ are given:

```
struct error_skb_list
{
   __u32 seq_num;
   __u32 rand_bit;
   int error_byte;
};
```

- The *seq_num* is the sequence number of the packet.
- The *rand_bit* field is the bit in the packet that is erred.

- The *error_byte* field is the original value of the byte in which the error is introduced.

The parameters related to ber qdisc are passed between user and kernel spaces in the format '*struct tc_ber_qopt*':

```
struct tc_ber_qopt
{
      int ber_value;          /* 1 error per ber_value bits */
};
```

- The term *ber_value* is the inverse bit error rate to be applied on the communicatin link.

5.2.1 Initialize

When a ber qdisc is created on an Ethernet device, the fields of '*struct ber_sched_data*' are initialized in *ber_init(struct Qdisc *sch, struct rtattr *opt)* implemented in net/sched/sch_ber.c. Through the document, the element *sch* is the '*struct Qdisc*' format of the ber qdisc and *opt* contains the parameters passed from the user space. The parameters are obtained in '*struct tc_ber _qopt*' element *ctl* as shown:

```
ctl = RTA_DATA(opt);
```

The value of ber passed from the user space is copied into '*struct ber_sched_data*' element *q*.

```
q->ber_value = ctl->ber_value;
```

5.2.2 Enqueue

When a packet is set for transmission by a device on which ber qdisc has been created, the *ber_enqueue((struct sk_buff *skb, struct Qdisc *sch)* routine is called by *dev_queue_xmit* to enqueue the packet into ber queue. The term *skb* represents the packet to be transmitted. The introduction of bit errors into packet flow is performed in *ber_enqueue( )*. The design to introduce BER is based on the following concepts:

- Since TCP/IP or UDP/IP protocol stacks are presently used for SBI emulation, ber qdisc is designed to introduce bit errors into packet flows with ETH_P_IP protocol and conforming to TCP or UDP transport layer protocols.

  Note: BER applied to packets belonging to other transport protocols may cause undesired results. Hence, other packet flows are transmitted without any interference.

- A packet given by '*struct sk_buff*' element *skb* has a length of *skb->len*. The first 68 bytes of *skb->len* are filled with network and transport layer information. The 68 bytes can be attributed to 20 bytes for transport layer header, 20 bytes for network layer header and 28 bytes transport layer data. Even if a bit error is introduced into these 68 bytes, there is no error introduced into the original packet because the header area is reconstructed each time a packet is transmitted. For further discussion, 68 bytes is referred to as HEADER_LEN and the total len of *skb->data* as MAX_LEN. No bit errors are introduced into the first HEADER_LEN area of *skb->data*, only the region between *skb->data* +

HEADER_LEN and *skb->data* + MAX_LEN is prone to errors in the BER emulation.

- Data transmitted over UDP transport layer protocol is unreliable. No retransmissions are requested or sent for loss of data. The sequence number on UDP packets is not unique in a packet flow. Therefore, the error inserted into UDP packets is not reset because the packet into which error is inserted is never retransmitted.
- The errors introduced into some packets in the error prone region in *skb->data* may not reflect in the retransmissions. If this is ignored, in the process of resetting errors, new errors are inserted into pure retransmission packets and the packet flow will be completely disrupted. Hence, when a retransmission for an erred packet is found, before resetting is performed, a check is made for the existence of the error.

The *struct ber_sched_data* object *q* associated with the device on which the ber qdisc is created is obtained from *sch* as:

```
q = (struct ber_sched_data *)(sch -> data);
```

5.2.2.1 Handling IP packets

On receiving a packet at *ber_enqueue( )*, a random number in the range 0 to *q->ber_value* is found using the random number generator *get_random( )* from C Standard library. *get_random( )* is chosen because it is fast and simple compared to other random number generators provided in the Linux kernel that take the system state for the generation.

get_random( ) gives a random value between 0 and (2^31 –1). To map the value to a range 0 to *q->ber_value*, the calculation performed is:

```
rand_bit = (unsigned) ( get_random( )  * q->ber_value )
                        ______________________________
                                  (2^31 -1)
```

If the *rand_bit* lies within the range *skb->data* + HEADER_LEN and *skb->data* + MAX_LEN, the following processing on the packet is done:

A new entry of type '*struct error_skb*' *err_skb_entry* is created for the packet.

- The field *seq_num* is copied from the TCP header of the packet if the packet belongs to TCP protocol. As explained earlier UDP protocol packet flows do not send retransmissions, hence they are not recorded. The *seq_num* for UDP entries is set to 0.
- The *rand_bit* field is filled with the random bit chosen for inserting error.

  ```
  err_skb_entry->rand_bit = rand_bit;
  ```
- The routine *err_reset( )* is called, the error is introduced and the field *error_byte* is set in *err_reset( )*.

5.2.2.2 Handling TCP retransmissions

Retransmits for TCP packets are erred because of the errors inserted into the original packet. Hence, they are reset to obtain the original packet. If the error is not reset, the packet will be rejected at the receiver and retransmits have to be sent again and again. The packet flow will not be complete. The process is described below:

When a packet is received, the transport layer protocol is determined.

- If the packet flow follows UDP, no further retransmission processing is performed since the UDP does not have retransmit sessions.
- If a TCP flow packet is received, it is discovered to be a retransmission if an entry already exists in the *err_list* with matching sequence number.
    - If the packet is not a retransmit, no further processing is performed.
    - If the packet is a retransmit, the routine *err_reset( )* is called with required parameters to reset the packet to original condition. The entry of type '*struct error_skb*' is then removed from the *err_list*.

Once the processing for normal and retransmit packets is performed, the packets are enqueued onto ber qdisc using *skb_queue_tail( )* routine defined in include/linux/skbuff.h.

```
skb_queue_tail(&sch->q, skb);
```

The backlog, bytes and packets fields of *sch* are updated.

On successful enqueue of the packet, a value of 0 is returned to the calling function.

5.2.3 Dequeue

The function *ber_dequeue(struct Qdisc *sch)* is invoked by *qdisc_restart( )* in net/sched/sch_generic.c to dequeue a packet from ber queue for transmission. The '*struct ber_sched_data*' element *q* associated with the device on which the ber qdisc is created is obtained from *sch* as follows:

```
q = (struct ber_sched_data *)sch->data;
```

A packet from ber queue is dequeued by calling *skb_dequeue( )* implemented in include/linux/skbuff.h.

```
skb_dequeue(&sch->q);
```

If the dequeue is successful, the packet dequeued is returned. If the dequeue fails, a value NULL is returned to the calling function.

5.2.4 Requeue

Once a packet is dequeued from ber queue, the *hard_start_xmit* of the device on which the ber qdisc is created is called. If the device is busy, the packet is requeued for later transmission. The *ber_requeue(struct sk_buff *skb, struct Qdisc *sch)* method is called to requeue a packet onto the ber queue. As executed in enqueuing of a packet, *skb_queue_head( )* is used for requeuing the packet *skb* on the ber queue.

```
skb_queue_head(&sch->q, skb);
```

5.2.5 Set and Reset error

To set or reset error in packets, *err_reset(struct sk_buff *skb, unsigned err_bit, int *err_byte_val, int orig_retx)* implemented in net/sched/sch_ber.c is called by *ber_enqueue( )*. *err_bit* is the bit in the data portion of the packet *skb*. *err_byte_val* is a pointer to value of the byte in which error is (to be) introduced. *orig_retx* is a flag to indicate if the packet is an original packet or a retransmit and is set to either ORIGINAL or RETX.

A brief description of the process is given:

change_byte is the address of byte in the packet that's set for change.

change_bit is a one byte variable with 1 as the change digit and 0s in the rest of the digits.

```
change_byte = skb->data + err_bit/8;
change_bit = (unsigned char)(128 >> (err_bit%8 - 1));
```

If the 6th bit in 159th byte of a packet, *change_byte* = *skb->data* + 158 and *change_bit* is (00000100)b.

- If *orig_retx* flag is set to RETX, the value of the error byte in the packet determined by *err_bit* is compared with the stored value at *err_byte_val*.
  - If the values match, the retransmit is determined the erred packet and the value at *err_bit* is flipped. The *err_byte_val* is set to the new value of *change_byte* and the control is returned.
  - If the values do not match, the retransmit does not have an error and is the original packet, so the control is returned with the packet untouched.
- If the *orig_retx* flag is set to ORIGINAL, the value at *err_bit* that was erred is flipped. The *err_byte_val* is set to the new value of *change_byte* and the control is returned.

5.2.6 Destroy

A qdisc is destroyed upon request from user space using '*tc*' command or when a system is set to reboot. This action is handled by *ber_destroy (struct Qdisc *sch)*. The module count of the ber module is decremented.

5.2.7 Modify attributes

A modification to the ber qdisc attributes is a change in the value of ber implemented on the packet flow transmitted by a device.

Upon request for a change from the user space, *ber_change(struct Qdisc *sch, struct rtattr *opt)* is called to change the value of *ber_value* field of '*struct ber_sched_data*'. The parameters from user space are obtained by converting *opt* to '*struct tc_ber_qopt*' format *ctl* using *RTA_DATA( )*. The new value of ber is copied.

```
q->ber_value = ctl->ber_value;
```

5.2.8 Display attributes

A request from user space to list the attributes of ber qdisc results in a call to *ber_dump(struct Qdisc *sch, struct sk_buff *skb)* routine. The attribute of ber qdisc, i.e., the value of bit error rate set to the packet flow on a device is returned in the format '*struct tc_ber_qopt*' *opt*. The element *opt* is passed to the user space.

```
opt.ber_value = ctl->ber_value;
```

5.2.9 Modifications to Linux kernel

The implementation of bit error rate queuing discipline is made in Linux kernel version 2.4.6. The parts of the kernel modified are listed:

- The definition of structure 'tc_ber_qopt' is made in include/linux/pkt_sched.h.

5.3 Iproute Interface

Iproute provides the interface to the queuing disciplines in Linux kernel. The command 'tc' is used to set the attributes of queuing disciplines. These attributes are passed into the kernel using netlink sockets. An interface for ber qdisc is created using Iproute. Ber qdisc can be created, modified, destroyed and it's attributes displayed using Iproute interface.

The format for setting the ber value of ber qdisc using *tc* command is:

```
tc qdisc add <DEV_HANDLE> ber ber_value INVERSE_OF_BER_VALUE
```

The format for changing the ber value associated with a ber qdisc on a device using *tc* command is:

```
tc     qdisc     change     <DEV_HANDLE>     ber     ber_value
INVERSE_OF_BER_VALUE
```

The format for destroying the ber qdisc using *tc* command is:

```
tc qdisc del <DEV_HANDLE> ber
```

The format for displaying the attributes of all qdiscs on the node using *tc* command is:

```
tc qdisc show
```

The parameters given using '*tc*' command are parsed using the routine *ber_parse_opt ( )*. The *ber_value* field in '*struct tc_ber_qopt*' element *qopt* is filled and passed into the kernel using netlink calls.

```
qopt.ber_value = (atoi)(*argv);
addattr_l(n, 1024, TCA_OPTIONS, &qopt, sizeof(qopt));
```

5.4 API for QoS interface

The API for QoS is used to dynamically set the QoS attributes of interfaces. A background on the API is given in the Background chapter. The API is extended to create, modify and delete *ber* queuing discipline through an interface.

The parameters required to process *ber* qdisc operations are in the format '*struct qdisc_ber*' similar to '*struct tc_ber_qopt*' for compatibility with QoS modules.

```
struct qdisc_ber
{
      int ber_value;
};
```

ber_value is the value of bit error rate to be set on the data transfer through the interface desired.

- To create, delete or modify a *ber* qdisc, an element *send_msg* of format '*struct qdisc_gen*' is filled with appropriate values. A pointer to *ber_opt* of type '*struct qdisc_ber*' is linked to it.

  ```
  ber_opt.ber_value = ber;
  send_msg.qdisc_options = (void *)&ber_opt;
  ```

- Using system call *set_qdisc*, the parameters are passed into kernel space. The operations are indicated using commands ADD_QDISC, DELETE_QDISC and CHANGE_QDISC. As a sample, creation operation of *ber* qdisc is shown.

  ```
  set_qdisc(ADD_QDISC,    (void    *)&send_msg,    sizeof(struct
  qdisc_gen));
  ```

- In net/qos_api/set_qdisc.c, a routine *fill_ber_params( )* is created to process API requests for *ber* qdisc. The user space arguments are processed and *usropt* of '*struct qdisc_ber*' format is filled.
- Using rtattr objects, internal kernel QoS modules are called for further processing of the queuing discipline.

```
fill_rtattr(&fill_param->nlm, 1024, TCA_OPTIONS, usropt,
sizeof(struct qdisc_ber));
```

The *ber* qdisc can thus be created, deleted or modified using the programming interface. This interface is used in the SBI emulation scenario to operate *ber* qdiscs on interfaces.

5.5 Conclusion

This chapter gives a detailed design for the introduction of bit errors into a packet flow. The Linux QoS modules are modified in the creation of ber queuing discipline. The implementation is extended to accommodate interfaces Iproute and API. The implementation is tested and the results are included in Chapter 8.

6 Incorporation of Delay and BER in SBI

Data propagation between a transmitter and a receiver is subject to fixed bandwidth, propagation delay and bit error rate. The designs for the emulation of propagation delay and bit error rate are provided in the previous two chapters. For the emulation of fixed bandwidth, Token Bucket Filter (*tbf*) is used. The emulation of communication links requires the emulation of the three properties.

'*Serialq*' is a hybrid qdisc that is designed to contain more than one qdisc in serial order. The packets passing through the '*serialq*' will be subject to all the inner qdiscs that the qdisc contains. In SBI emulation, *serialq* is created using Token Bucket Filter (*tbf*), Propagation Delay (*delay*) and Bit Error Rate (*ber*) qdiscs in that order. Hence, a packet flow through the device on which '*serialq*' is installed is subject to *tbf*, *delay* and *ber* qdisc properties. The design for *serialq* is explained in this chapter.

6.1 Design Overview

Serialq is implemented as a qdisc in Linux kernel. The functioning of a *serialq* qdisc depends on the presence of component qdiscs on the same device as the *serialq* qdisc. More than one qdisc can be created on a single device using CBQ. The required background on CBQ is provided in Chapter 2.

The qdiscs *serialq*, *tbf*, *delay* and *ber* are created on separate classes on an Ethernet device using CBQ. All the traffic passing through the device is directed to the class containing *serialq* qdisc using filters. The direction of packet flow to *serialq* qdisc is shown in Figure 11. The *serialq* qdisc contains pointers to the *tbf*, *delay* and *ber* qdiscs on the same device.

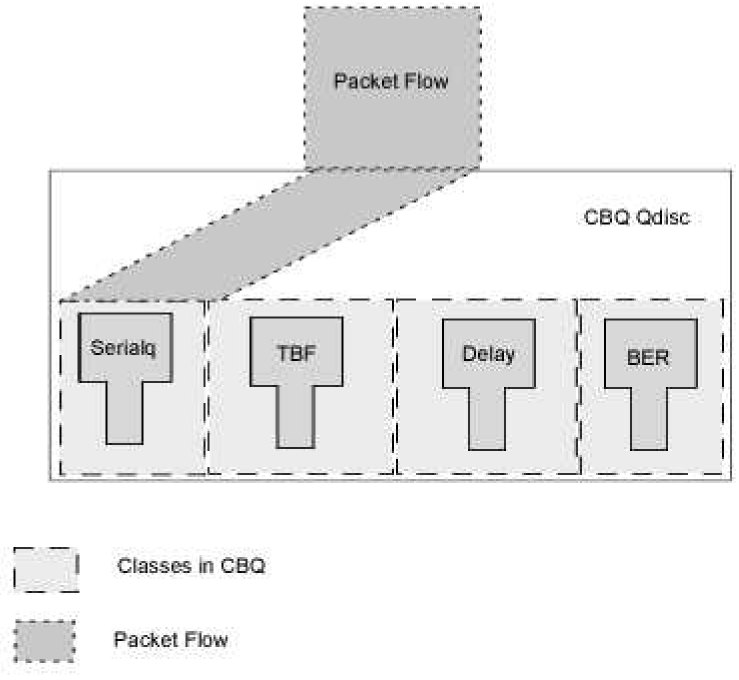


Figure 11: Serialq and Component Queuing Disciplines

The design for packet flow through *serialq* qdisc is shown in Figure 12. On enqueuing a packet on an Ethernet device that has *serialq* qdisc, it is directed to *serialq* qdisc using appropriate filters. The packet flows through the component qdiscs of the *serialq* during the enqueue and dequeue process.

The enqueue routine on *serialq* enqueues the packet on *tbf* queue, dequeues it from *tbf* queue and then enqueues it on *delay* qdisc. The dequeue routine on *serialq* dequeues a packet from *delay* queue, enqueues on and dequeues from *ber* queue. A packet flow is thus set to follow the properties of *tbf*, *delay* and *ber* queuing disciplines by the use of *serialq* qdisc.

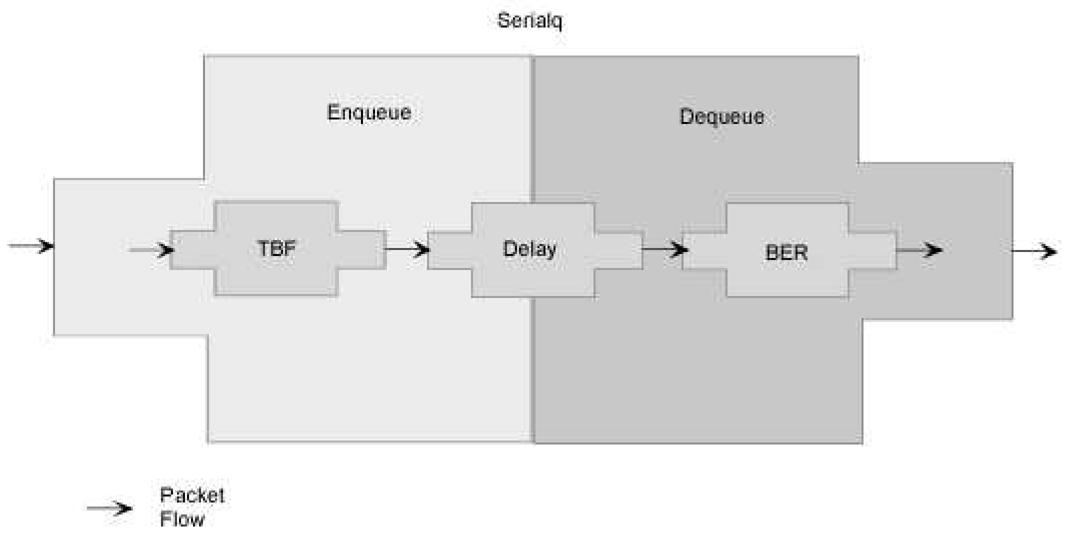


Figure 12: Packet Flow Through a Serialq Queuing Discipline

6.2 Serialq in Linux Kernel QoS

An Ethernet device queues packets for transmission using a queuing discipline (qdisc) in Linux kernel QoS. *Serialq* qdisc is created to merge the properties of *tbf*, *delay* and *ber* qdiscs into packet flows through the device on which it is created.

When a qdisc is created in Linux Kernel QoS, elements of formats '*struct Qdisc*' and '*struct Qdisc_ops*' are associated with it. For the '*serialq*' qdisc, the elements of '*struct Qdisc_ops*' used to perform operations on the qdisc are: *serialq_init*, *serialq_enqueue*, *serialq_dequeue*, *serialq_requeue*, *serialq_change* and *serialq_destroy*.

The details of '*struct serialq_sched_data*' that contains the attributes of *serialq* qdisc:

```
struct serialq_sched_data
{
      struct Qdisc *tbf_qdisc;
      struct Qdisc *delay_qdisc;
      struct Qdisc *ber_qdisc;
};
```

- The *tbf_qdisc* field is the link associated with the component *tbf* qdisc.
- The *delay_qdisc* field is the link associated with the component *delay* qdisc.
- The *ber_qdisc* field is the link associated with the component *ber* qdisc.

The functioning of *serialq* relies on the existence of atleast one component qdisc. The design for *serialq* allows the absence of *tbf* and *ber* qdiscs, so a *serialq* cannot be complete without *delay* qdisc. *Delay* qdisc is arbitrarily chosen are mandatory.

6.2.1 Initialize

The initialization of *serialq* qdisc processes in two steps:

- Creation of *serialq* qdisc
- Creation of *tbf*, *delay* and *ber* qdiscs

6.2.1.1 Serialq qdisc

When a *serialq* qdisc is created on an Ethernet device, the fields of '*struct serialq_sched_data*' are initialized in *serialq_init(struct Qdisc *sch, struct rtattr *opt)* implemented in net/sched/sch_serialq.c. The element *sch* is the '*struct Qdisc*' format of the *serialq* qdisc and *opt* contains the parameters passed from the user space. The parameters are obtained in '*struct tc_serialq_qopt*' element *ctl* from *opt*:

The other elements of '*struct serialq_sched_data' q*, *tbf_qdisc*, *delay_qdisc*, *ber_qdisc* are set to NULL in the process initiated by the creation of *serialq* qdisc.

6.2.1.2 Component qdiscs

The fields *tbf_qdisc*, *delay_qdisc* and *ber_qdisc* in '*struct serialq_sched_data*' element *sq* are filled when the component qdiscs - *tbf*, *delay* and *ber* are created. The process is implemented in *cbq_graft( )* in net/sched/sch_cbq.c.

When a qdisc is attached to a class on *cbq* qdisc, *cbq_graft( )* is called. Once the *serialq* qdisc is attached to a class, every subsequent qdisc attached to one of the classes on the device is used to set the links in *sq*.

In detail, when a *tbf* qdisc is attached after *serialq* qdisc is attached on the same device, the *tbf_qdisc* link in *sq*, if not yet set, is set to the '*struct Qdisc*' variable *new* associated with *tbf* qdisc.

```
if(!(sq->tbf_qdisc))
{
      if(!strcmp(new->ops->id, "tbf"))
      sq->tbf_qdisc = new;
}
```

The *delay* and *ber* qdisc links are also set in *cbq_graft()*.

6.2.2 Enqueue

The process of packet transmission on a *serialq* qdisc is provided in the previous sections. The design details are given in this section.

When a packet is set for transmission by a device on which the setup for *serialq* qdisc is created, the *cbq_enqueue(struct sk_buff *skb, struct Qdisc *sch)* is called by *dev_queue_xmit( )* in net/core/dev.c. The *cbq_enqueue( )* inturn calls the *serialq_enqueue(struct sk_buff *skb, struct Qdisc *sch)* routine to enqueue the packet. The term *skb* represents the packet to be transmitted.

The '*struct serialq_sched_data*' object *q* associated with the device on which the *serialq* qdisc is created is obtained from *sch* as:

```
q = (struct serialq_sched_data *)(sch -> data);
```

The design to propagate the packet through the *serialq* components – *tbf*, *delay*, *ber* qdiscs is provided below.

If no *delay* qdisc is associated with the *serialq* qdisc, the packet is dropped and a return value NET_XMIT_DROP is returned. The presence of *delay* qdisc is essential for the functioning of *serialq* qdisc.

If a *tbf* qdisc is associated with the *serialq*, the *skb* is enqueued onto *tbf* qdisc.

```
if(q->tbf_qdisc)
      q->tbf_qdisc->enqueue(skb, q->tbf_qdisc);
```

On success, the packet is dequeued from *tbf* qdisc.

```
tbf_skb = q->tbf_qdisc->dequeue(q->tbf_qdisc);
```

If *tbf* dequeue returns a packet, the packet is enqueued onto *delay* queue.

On success, the backlog, bytes and packets fields of *sch* are updated. In case of any errors, *serialq_enqueue( )* returns NET_XMIT_DROP.

6.2.3 Dequeue

The function *cbq_dequeue(struct Qdisc *sch)* is invoked by *qdisc_restart( )* in net/sched/sch_generic.c to dequeue a packet from the queues on the device on which *serialq* is created. s*erialq_dequeue(struct Qdisc *sch)* is called by *cbq_dequeue( )* as the packet flow passes through the class containing *serialq* qdisc. The '*struct serialq_sched_data*' element *q* associated with the device on which the *serialq* qdisc is created is obtained from *sch* as follows:

```
q = (struct serialq_sched_data *)sch->data;
```

A packet is dequeued from the *delay* qdisc attached to *serialq*. If the *delay* dequeue fails, NULL is returned to the calling function. On success, the packet dequeued is queued onto and dequeued from the *ber* qdisc. The packet dequeued from *ber* qdisc is returned to the calling function. In case of any errors from dequeue routines, *serialq_dequeue( )* returns a NULL value. In case of errors from ber enqueue routine, the packet dequeued from delay queue is returned.

6.2.4 Requeue

Once a packet is dequeued from serialq queue, the *hard_start_xmit* of the device on which the serialq qdisc is created is called. If the device is busy, the packet is requeued for later transmission. The *serialq_requeue(struct sk_buff *skb, struct Qdisc*

**sch)* method is called to requeue a packet onto the serialq queue. The routine *serialq_enqueue( )* is called to re-enqueue the packet onto serialq components.

```
serialq_enqueue(skb, sch);
```

6.2.5 Destroy

A qdisc is destroyed upon request from user space using '*tc*' command or when a system is set to reboot. This action is handled by *serialq_destroy (struct Qdisc *sch)*. The links to the other qdiscs – *tbf*, *delay*, *ber* are removed and the module count of the *serialq* module is decremented.

6.2.6 Display attributes

A request from user space to list the attributes of *serialq* qdisc results in a call to *serialq_dump(struct Qdisc *sch, struct sk_buff *skb)* routine. The attribute of *serialq* qdisc on display is the device name and is obtained from *sch->dev->name*. The information is returned in the format '*struct tc_serialq_qopt*' *opt*. The *opt* is passed to the user space.

```
strncpy(opt.dev_name, sch->dev->name, IFNAMSIZ);
```

6.2.7 Modifications to Linux kernel

The implementation of serialq queuing discipline is made in Linux kernel version 2.4.6. The parts of the kernel modified are listed:

- The definition of structure 'tc_serialq_qopt' is made in include/linux/pkt_sched.h.
- The initialization of serialq qdisc is set in net/sched/sch_api.c

- CBQ implementation is modified in cbq_graft( ) and cbq_dequeue_prio( ) in net/sched/sch_cbq.c

6.3 Iproute Interface

Iproute provides the interface to the queuing disciplines in Linux kernel. The command ‘tc’ is used to set the attributes of queuing disciplines. These attributes are passed into the kernel using netlink sockets. An interface for *serialq* qdisc is created using Iproute. *Serialq* qdisc can be created, modified, destroyed and displayed using Iproute interface.

The format for creating a *serialq* qdisc using *tc* command is:

```
tc qdisc add <DEV_HANDLE> serialq
```

The element DEV_HANDLE contains the generic fields related to device on which the qdisc is created and handles to the qdisc. This is defined in Chapter 2.

The format for destroying the *serialq* qdisc using *tc* command is:

```
tc qdisc del <DEV_HANDLE> serialq
```

The format for displaying all qdiscs on the node using *tc* command is:

```
tc qdisc show
```

The parameters obtained from the ‘*tc*’ command are used to fill in the field in ‘*struct tc_serialq_qopt*’:

```
struct tc_serialq_qopt
{
      char dev_name[16];
};
```

The *dev_name* field is nominal and does not carry any significance in the serialq setup.

The parameters given using '*tc*' command are parsed using the routine *serialq_parse_opt ( )*. The *dev_name* field in '*struct tc_serialq_qopt*' element is filled and passed into the kernel using netlink calls.

```
strncpy(opt.dev_name, *argv, IFNAMSIZ)
addattr_l(n, 1024, TCA_OPTIONS, &qopt, sizeof(qopt));
```

The process for creating a *serialq* qdisc involves the following procedure:

- Create a cbq qdisc on the device on which *serialq* is desired
- Create classes one each for *serialq*, *tbf*, *delay* and *ber* qdiscs on the cbq qdisc
- Create *serialq*, *tbf*, *delay* and *ber* qdiscs on the classes
- Create a filter that directs all the traffic on the device to the class on which *serialq* is created

6.4 API for QoS interface

The API for QoS is used to dynamically set the QoS attributes of interfaces. A background on the API is given in the Chapter 2. The API is extended to create, modify and delete *serialq* queuing discipline through an interface.

The parameters required to process *serialq* qdisc operations are in the format '*struct qdisc_serialq*'.

```
struct qdisc_serialq
```

```
{
      char dev_name[IFNAMSIZ];
};
```

dev_name is the Ethernet device on which the serialq is to be created.

- To create, delete or modify a *serialq* qdisc, an element *send_msg* of format '*struct qdisc_gen*' is filled with appropriate values. A pointer to *serialq_opt* of type '*struct qdisc_serialq*' is linked to it.

  ```
  strcpy(serialq_opt.dev_name, intfName);
  send_msg.qdisc_options = (void *)&serialq_opt;
  ```

- Using system call *set_qdisc*, the parameters are passed into kernel space. The operations are indicated using commands ADD_QDISC, DELETE_QDISC and CHANGE_QDISC. As a sample, creation operation of *serialq* qdisc is shown.

  ```
  set_qdisc(ADD_QDISC,   (void   *)&send_msg,   sizeof(struct
  qdisc_gen));
  ```

- In net/qos_api/set_qdisc.c, a routine *fill_serialq_params( )* is created to process API requests for *serialq* qdisc. The user space arguments are processed and *usropt* of '*struct qdisc_serialq*' format is filled.

- Using rtattr objects, internal kernel QoS modules are called for further processing of the qdisc.

  ```
  fill_rtattr(&fill_param->nlm,  1024,  TCA_OPTIONS,  usropt,
  sizeof(struct qdisc_serialq));
  ```

The *serialq* qdisc can thus be created, deleted or modified using the programming interface. This interface is used in the SBI emulation scenario to operate *serialq* qdiscs on interfaces.

6.5 Conclusion

This chapter gives a detailed design for the creation of a unique queuing discipline that combines more than one qdisc in serial order. The serialq qdisc implementation results in providing an effective of tbf, delay and ber qdiscs on a packet flow. The implementation is extended for interfaces Iproute and API. The implementation is tested and the results are included in Chapter 8.

7 Emulation of a SBI Communication Link

The designs for Ethernet device emulation and the introduction of communication link properties into packet flow are provided in the previous chapters. This chapter gives the integration of Ethernet device and QoS emulation into the SBI emulation system.

7.1 Virtual Devices

Virtual Ethernet Devices are created and destroyed by the EM Node Controller on starting and close of an emulation scenario. The API created in the Device design is used to perform these operations from the Controller.

7.1.1.1 Start

On starting the emulation scenario, the EM Node Controller creates Veth devices on a Node. The EM Node Controller accesses the Linux kernel using ioctl calls to create Veth devices.

A PF_INET socket is opened to perform the ioctl call operation. An element *param_create* of format '*struct veth_param_create*' is filled.

The field *phyDevName* is the Physical Ethernet device on which the Veth device has to be created and the *srcMac* field is the MAC address of the Veth device. VETHDEV_CREATE is the command used to indicate the creation of a Veth device. With the required parameters, the ioctl call is made to the kernel to create the device.

```
ioctl(ioctl_fd, VETHDEV_CREATE, (char *)&param_create);
```

On successful creation, the interface number of the Veth device is returned in *itfNum* field in *param_create*.

7.1.1.2 Terminate

On receiving a signal to stop the SBI emulation scenario, the Veth devices on a Node are destroyed. Each device is destroyed on the device using ioctl calls.

The process is the same as the creation of a Veth device. An element *param_destroy* of format '*struct veth_param_destroy*' is filled. The *itfNum* field is the interface number of the Veth device to be destroyed. VETHDEV_DESTROY is the command used to indicate the deletion of a Veth device. With the required parameters, the ioctl call is made to the kernel to destroy the device. On successful deletion, a value 0 is returned.

```
ioctl(ioctl_fd, VETHDEV_DESTROY, (char *)&param_destroy);
```

The EM Node Controller creates and destroys the Veth devices in SBI emulation environment on receiving Start and Stop messages from the EM. The Veth devices are used as interfaces to transfer data between Nodes.

7.2 QoS Features

On startup, the EM Node Controller sets the QoS attributes bandwidth, propagation delay and bit error rate are set on each interface on a Node. The parameters bandwidth, the amount of propagation delay and the bit error rate value on each interface are passed from the EM to the Node. An API for setting QoS attributes on Ethernet devices is used to set the link parameters. The details about the API for Linux QoS can be referred to in Chapter 2.

7.2.1 Create

The QoS setup on interfaces on a Node requires the creation of a set of qdiscs, classes and a filter. The QoS elements are created using the QoS API. The procedure follows:

- Create a cbq qdisc on each desired interface with required parameters.
- Create four classes one each for *serialq*, *tbf*, *delay* and *ber* qdiscs on the cbq qdisc.
- Create *serialq* qdisc on one of the classes.

  The fields *dev* in '*struct qdisc_gen*' element *send_msg* and *dev_name* in '*struct qdisc_serialq*' element *serialq_opt* are filled with the interface name. The parent and the qdisc type 'serialq' are set in *send_msg*. The pointer to *serialq_opt* is set in *send_msg* and the *set_qdisc( )* routine is used to create the qdisc in the kernel on the desired interface.

  ```
  send_msg.qdisc_options = (void *)&serialq_opt;
  set_qdisc(ADD_QDISC,   (void   *)&send_msg,   sizeof(struct
  qdisc_gen));
  ```

- Create *tbf* qdisc on the second class.

  The field *dev* in '*struct qdisc_gen*' element *send_msg* is filled with the interface name. The parent, limit, burst and bandwidth and the qdisc type 'tbf' are set in '*struct qdisc_tbf_usr*' element *tbf_opt*. The pointer to *tbf_opt* is set in *send_msg* and the *set_qdisc( )* routine is used to create the tbf qdisc in the kernel on the desired interface.

```
send_msg.qdisc_options = (void *)&tbf_opt;
set_qdisc(ADD_QDISC,   (void   *)&send_msg,   sizeof(struct
qdisc_gen));
```

- Create *delay* qdisc on one of the classes.

  The field *dev* in '*struct qdisc_gen*' element *send_msg* is filled with the interface name. The parent and the qdisc type 'delay' are set in *send_msg*. The *delay_msec* field in '*struct qdisc_delay*' element *delay_opt* is set to the propagation delay value passed from the EM. The pointer to *delay_opt* is set in *send_msg* and the *set_qdisc( )* routine is used to create the delay qdisc in the kernel on the desired interface.

```
send_msg.qdisc_options = (void *)&delay_opt;
set_qdisc(ADD_QDISC,   (void   *)&send_msg,   sizeof(struct
qdisc_gen));
```

- Create *ber* qdisc on one of the classes.

  The field *dev* in '*struct qdisc_gen*' element *send_msg* is filled with the interface name. The parent and the qdisc type 'ber' are set in *send_msg*. The *ber_value* field in '*struct qdisc_ber*' element *ber_opt* is set to the bit error rate value passed from the EM. The pointer to *ber_opt* is set in *send_msg* and the *set_qdisc( )* routine is used to create the qdisc in the kernel on the desired interface.

```
send_msg.qdisc_options = (void *)&ber_opt;
```

```
set_qdisc(ADD_QDISC,    (void   *)&send_msg,    sizeof(struct
qdisc_gen));
```

- Create a filter that directs all the traffic on the device to the class on which *serialq* is created.

This completes the QoS setup on an interface.

7.2.2 Modify

The communication link parameters can be modified in the emulation system to suit the changing links between interfaces. The parameters that can be modified are the propagation delay value and the bit error rate on an interface. The process is the same as described in the Create section. The option 'CHANGE_QDISC' is used in the *set_qdisc( )* routine for the qdisc to be modified.

7.2.3 Destroy

On the close of the emulatin scenario, the QoS settings on each interface are removed. The *cbq* qdisc created is destroyed using the API with delete option 'DELETE_QDISC' is used in *set_qdisc( )* routine.

7.3 Conclusion

This chapter gives the details of emulation of communication link and properties in the SBI emulation system. The SBI Node emulation system is tested and the resulting setup of Virtual Ethernet devices and the QoS attributes on the devices are shown in Chapter 8.

8 Testing and Results

The designs provided in the previous chapters are tested. The results are shown in this chapter.

8.1 Virtual Ethernet Device Configuration using Vethctl

Virtual Ethernet Devices can be created, deleted and listed using the interface Vethctl. The commands used and the results obtained are shown:

```
bluenode01 [2] # Vethctl -c eth1 00:93:93:30:02:02
vethctl: Virtual Device veth0 created successfully

bluenode01 [3] # ifconfig veth0 10.67.20.1
bluenode01 [4] # Vethctl -c eth1 00:93:92:30:52:02
vethctl: Virtual Device veth1 created successfully

bluenode01 [5] # ifconfig veth1 10.67.21.1
bluenode01 [6] # ifconfig eth1 192.168.10.21
bluenode01 [7] # Vethctl -l
Number of veth devices on this host: 2

List of devices:
On eth1 :    2 virtual devices

   Virtual device   Physical device   itfNum     Mac address
       veth1            eth1             1         00:93:92:30:52:02
       veth0            eth1             0         00:93:93:30:02:02
```

```
bluenode01 [8] # ifconfig veth0
veth0     Link encap:Ethernet  HWaddr 00:93:93:30:02:02
          inet addr:10.67.20.1  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100

bluenode01 [9] # ifconfig veth1
veth1     Link encap:Ethernet  HWaddr 00:93:92:30:52:02
          inet addr:10.67.21.1  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100

bluenode01 [10] # ifconfig veth1 down
bluenode01 [11] # Vethctl -d 1
vethctl: Virtual Device veth1 destroyed successfully

bluenode01 [12] # Vethctl -l
Number of veth devices on this host: 1

List of devices:
On eth1 :    1 virtual devices

   Virtual device   Physical device   itfNum     Mac address
       veth0             eth1             0         00:93:93:30:02:02

bluenode01 [13] # ifconfig veth0 down
bluenode01 [14] # Vethctl -d 0
vethctl: Virtual Device veth0 destroyed successfully

bluenode01 [15] # Vethctl -l
```

```
Number of veth devices on this host: 0
```

8.2 Queuing discipline configuration using ‘tc’

The delay, ber and serialq queuing disciplines can be managed using Iproute ‘tc’ interface. A sample configuration is given in Appendix B. The results are shown here:

```
bluenode01 [1] # tc qdisc add dev eth1 root handle 1 cbq bandwidth
10mbps avpkt 1200
bluenode01 [2] #
bluenode01 [2] # tc class add dev eth1 parent 1: classid 1:1 est
1sec 8sec cbq bandwidth 80Mbit 15mbps allot 1514 maxburst 50 avpkt
1000 prio 1
bluenode01 [3] # tc class add dev eth1 parent 1: classid 1:2 est
1sec 8sec cbq bandwidth 1Mbit 15mbps allot 1514 maxburst 50 avpkt
1000 prio 8
bluenode01 [4] # tc class add dev eth1 parent 1: classid 1:3 est
1sec 8sec cbq bandwidth 1Mbit 1mbps allot 1514 maxburst 50 avpkt
1000 prio 8
bluenode01 [5] # tc class add dev eth1 parent 1: classid 1:4 est
1sec 8sec cbq bandwidth 1Mbit 1mbps allot 1514 maxburst 50 avpkt
1000 prio 8
bluenode01 [6] #
bluenode01 [6] # tc qdisc add dev eth1 parent 1:1 handle 4 serialq
dev_name eth1
bluenode01 [7] # tc qdisc add dev eth1 parent 1:2 handle 5 delay
delay_msec 20
```

```
bluenode01 [8] # tc qdisc add dev eth1 parent 1:3 handle 6 tbf limit
120000 burst 150000 rate 1
bluenode01 [9] # tc qdisc add dev eth1 parent 1:4 handle 7 ber
ber_value 10000000
bluenode01 [10] #
bluenode01 [10] # tc filter add dev eth1 pref 10 protocol ip u32
match ip protocol 0 0x00 class1
bluenode01 [11] #
bluenode01 [11] # tc qdisc show
qdisc ber 7: dev eth1 ber_value: 10000000
qdisc tbf 6: dev eth1 rate 80Mbit burst 146790b lat 4292.4s
qdisc delay 5: dev eth1 delay_value: 20
qdisc serialq 4: dev eth1
qdisc cbq 1: dev eth1 rate 80Mbit (bounded,isolated) prio no-
transmit
bluenode01 [12] #
bluenode01 [12] # tc class show dev eth1
class cbq 1: root rate 80Mbit (bounded,isolated) prio no-transmit
class cbq 1:1 parent 1: leaf 4: rate 120Mbit prio 1
class cbq 1:2 parent 1: leaf 5: rate 120Mbit prio no-transmit
class cbq 1:3 parent 1: leaf 6: rate 8Mbit prio no-transmit
class cbq 1:4 parent 1: leaf 7: rate 8Mbit prio no-transmit
bluenode01 [13] #
bluenode01 [13] # tc filter show dev eth1
filter parent 1: protocol ip pref 10 u32
filter parent 1: protocol ip pref 10 u32 fh 800: ht divisor 1
filter parent 1: protocol ip pref 10 u32 fh 800::800 order 2048 key
ht 800 bkt 0 flowid 1:1
  match 00000000/00000000 at 8
```

The results show that the delay, ber and serialq qdiscs are created using ‘tc’ and the parameters are listed using the operation ‘*tc qdisc show*’.

8.3 Configuration in SBI Emulation

In SBI emulation, Veth devices are created using ioctl calls by the EM Node Controller in the SBI Node. The queuing disciplines are created using API developed for the delay, ber and serialq queuing disciplines. U32 filters are used in the QoS setup on the interfaces.

The output of a SBI emulation in the fields of Veth device creation and setup of QoS on Veth devices is shown:

ifconfig results:

```
veth0     Link encap:Ethernet  HWaddr 00:04:86:00:08:00
          inet addr:10.67.8.1  Bcast:10.255.255.255
Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100

veth1     Link encap:Ethernet  HWaddr 00:04:86:00:08:01
          inet addr:10.67.8.2  Bcast:10.255.255.255
Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100

veth2     Link encap:Ethernet  HWaddr 00:04:86:00:08:02
          inet addr:10.67.8.3  Bcast:10.255.255.255
Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100

veth3     Link encap:Ethernet  HWaddr 00:04:86:00:08:03
          inet addr:10.67.8.4  Bcast:10.255.255.255
Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
```

8.3.1.1 Vethctl –l:

```
Number of veth devices on this host: 4

List of devices:
On eth1 :    4 virtual devices

   Virtual device   Physical device   itfNum     Mac address
       veth3             eth1              3
00:04:86:00:08:03
       veth2             eth1              2
00:04:86:00:08:02
       veth1             eth1              1
00:04:86:00:08:01
       veth0             eth1              0
00:04:86:00:08:00
```

8.3.1.2 tc qdisc show:

```
qdisc ber 8004: dev veth0 ber_value: 0
qdisc tbf 8003: dev veth0 rate 80Kbit burst 149995b lat 2147.5s
qdisc delay 8002: dev veth0 delay_value: 0
qdisc serialq 8001: dev veth0
qdisc cbq 1: dev veth0 rate 80Mbit (bounded,isolated) prio no-
transmit
qdisc ber 8008: dev veth1 ber_value: 1259902592
qdisc tbf 8007: dev veth1 rate 80Kbit burst 149995b lat 2147.5s
qdisc delay 8006: dev veth1 delay_value: 45
qdisc serialq 8005: dev veth1
qdisc cbq 2: dev veth1 rate 80Mbit (bounded,isolated) prio no-
transmit
qdisc ber 800c: dev veth2 ber_value: 1259902592
qdisc tbf 800b: dev veth2 rate 80Kbit burst 149995b lat 2147.5s
qdisc delay 800a: dev veth2 delay_value: 12
qdisc serialq 8009: dev veth2
qdisc cbq 3: dev veth2 rate 80Mbit (bounded,isolated) prio no-
transmit
qdisc ber 8010: dev veth3 ber_value: 1259902592
qdisc tbf 800f: dev veth3 rate 80Kbit burst 149995b lat 2147.5s
qdisc delay 800e: dev veth3 delay_value: 45
qdisc serialq 800d: dev veth3
qdisc cbq 4: dev veth3 rate 80Mbit (bounded,isolated) prio no-
transmit
```

8.3.1.3 tc class show dev veth0:

```
class cbq 1: root rate 80Mbit (bounded,isolated) prio no-transmit
class cbq 1:1 parent 1: leaf 8001: rate 1Mbit prio 1
class cbq 1:2 parent 1: leaf 8002: rate 1Mbit prio 1
class cbq 1:3 parent 1: leaf 8003: rate 1Mbit prio 1
class cbq 1:4 parent 1: leaf 8004: rate 1Mbit prio 1
```

8.3.1.4 tc filter show dev veth0:

```
filter parent 1: protocol ip pref 10 u32
filter parent 1: protocol ip pref 10 u32 fh 800: ht divisor 1
filter parent 1: protocol ip pref 10 u32 fh 800::800 order 2048 key
ht 800 bkt 0 flowid 1:1
  match 00000000/00000000 at 8
```

8.4 Performance Results for Virtual Ethernet Devices

The performance of Virtual Ethernet devices is tested by the following setup.

On two separate machines, simultaneous testing on the devices is performed under the following scenarios:

- No veth devices
- A single Veth device on each machine
- Two Veth devices on each machine
- Three Veth devices on each machine

The results are shown:

Number of Veth device On Each Node	Device of transfer	Effective Throughput (in Mbps)
0	eth1	89.74
1	veth0	89.42
2	veth0	44.26
	veth1	44.20
3	veth0	29.28
	veth1	29.24
	veth2	29.01

It can be seen that during simultaneous transmission on more than on Virtual Ethernet devices, the bandwidth of the underlying Physical Ethernet device gets divided among them.

8.5 Performance Results for Delay Queuing Discipline

The performance of Delay queuing discipline is tested by varying the value of delay and comparing with the average delay of 200 ping packets.

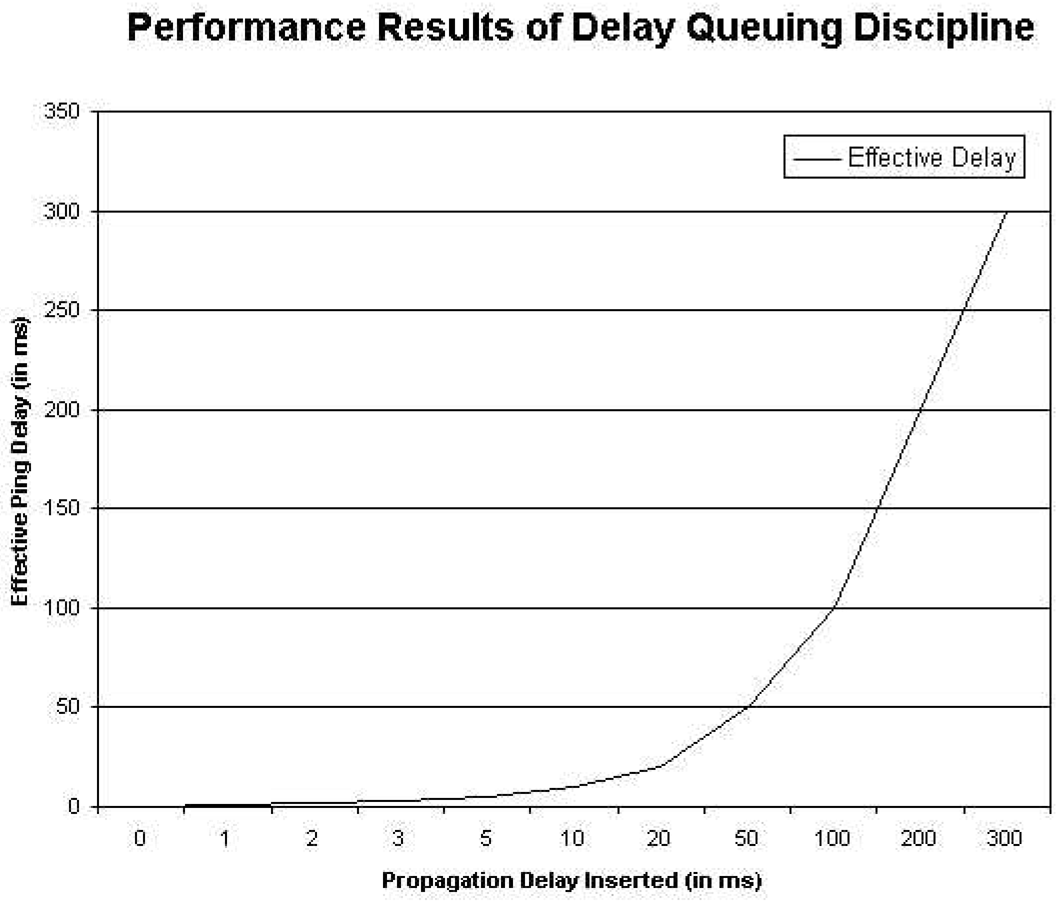


Figure 13: Performance of Propagation Delay Queuing Discipline

The results given in Figure 13 show that the effective propagation delay is equal to the amount of delay inserted into a packet flow.

8.6 Performance Results for BER Queuing Discipline

The performance of Bit Error Rate queuing discipline is evaluated in this section. A BER queuing discipline is created on a Veth device. The effect of BER on the throughput and response times of ttcp generated packet flows are observed. On one of the test nodes, BER value is varied from 10e-8 to 10e-5. Typical number of error packets at any time does not exceed 60 for BER upto 10e-6. For higher BERs, error packets also increase.

As seen in Figure 14, the throughput reduces as the BER increases in a packet flow. Also, the response times for packet flow increase as BER increases due to increased TCP retransmissions required to complete the flow. The application ‘ttcp’ is run with packet flows of 10e5 packets with length 1000 bytes each. The results are shown in the following graph.

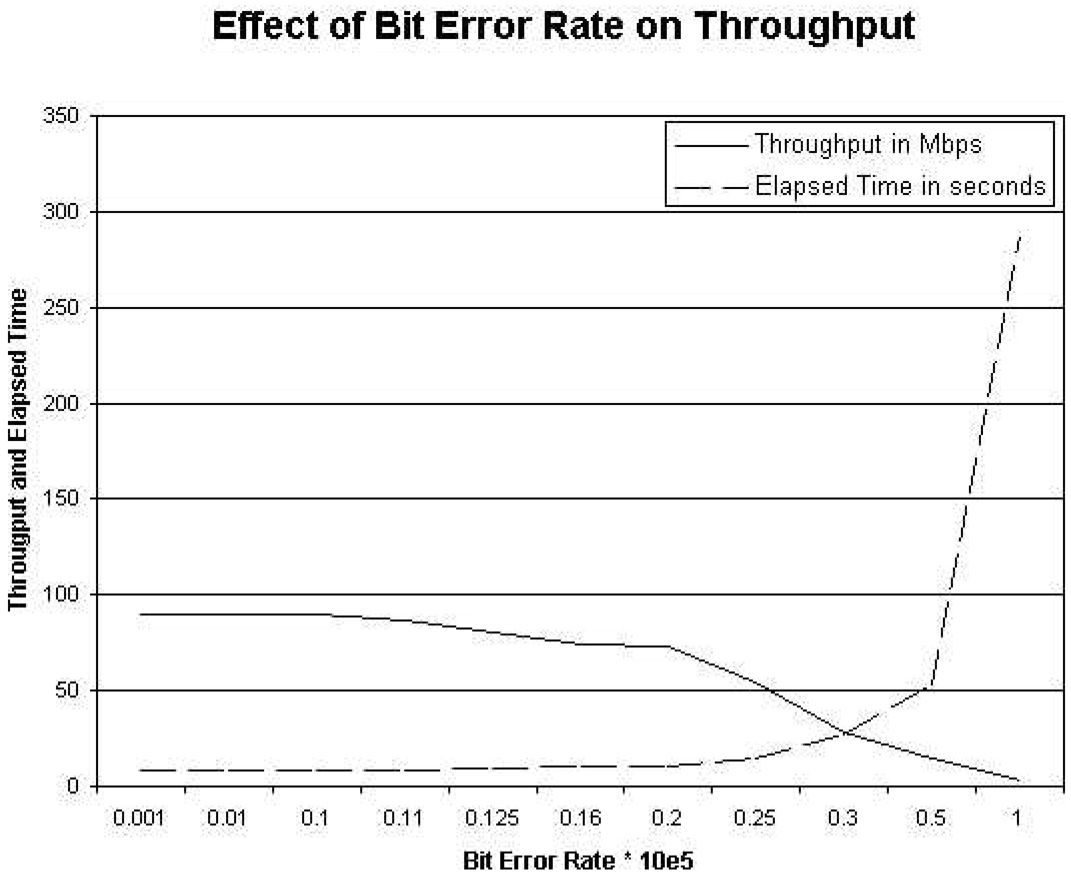


Figure 14: Performance of Bit Error Rate Queuing Discipline

9 Conclusions

This thesis work provides the detailed design for Ethernet device emulation. It also describes the design of Propagation Delay, BER and Serialq queuing disciplines that contribute significantly to the Linux QoS. This provides a complete design for the emulation of a communication link in the SBI emulation system.

9.1 Future Work

Possible extensions to the work given in this document include:

- BER queueing discipline could be extended to introduce errors into other transport layer protocols. Presently, only UDP and TCP protocols are erred.
- Serialq has no control over the management of it's component queuing disciplines. The design can be modified so that the creation and deletion of the components is controlled by serialq.
- Better handling of component tbf qdisc dequeue is required in Serialq implementation. If tbf dequeue fails, packet cannot be freed because it is stuck in the tbf queue. Presently, it is enqueued onto delay queue. The pointers in skb get overwritten due to this.

10 References

[1] Bruce R. Elbert, *The Satellite Communication Applications Handbook*, Norwood, MA: Artech House, 1997, pp 7–20.

[2] Yurong Hu and Victor O. K. Li, The University of Hong Kong, 'Satellite-Based Internet: A Tutorial', IEEE Communications Magazine, Vol. 39, No. 3, March 2001, pp 154-162,

[3] Lloyd Wood, George Pavlou, and Barry Evans, University of Surrey, 'Effects on TCP of Routing Strategies in Satellite Constellations', IEEE Communications Magazine, Vol. 39, No. 3, March 2001, pp 172-180.

[4] Dennis Roddy, *Satellite Communications*, Second Edition, McGraw-hill, 1996, pp 89-99, 279-296.

[5] Eylem Ekici, Ian F. Akyildiz, Michael D. Bender, 'A Distributed Routing Algorithm for Datagram Traffic in LEO Satellite Networks', IEEE/ACM Transactions On Networking, Vol. 9, No. 2, April 2001, pp 137-138.

[6] James Martin, *Communications Satellite Systems*, Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1978, pp 90-130, 257-276.

[7] Thomas R. Henderson and Randy H. Katz, University of California, Berkeley, 'Network Simulation For LEO Satellite Networks', AIAA Paper 2000-1237, 1-3.

[8] Alexander Keller, Munich University of Technology, 'Towards CORBA-based Enterprise Management: Managing CORBA-based Systems with SNMP Platforms', Proceedings of the Second International Enterprise Distributed Object Computing Workshop, EDOC'98, San Diego, CA, USA: Munich Network Management Team, November 1998.

[9] Saleem N. Bhatti, http://www.cs.ucl.ac.uk/staff/S.Bhatti/D51-notes/notes.html, October 1994

[10] Christy Hudgins-Bonafield, http://www.networkcomputing.com/905/905f2.html

[11] Jonathan Angel, http://www.networkmagazine.com/article/NMG20000509S0033

[12] Leon S. Searl, 'Space Based Internet System Architecture', February 22, 2001, pp 4-11.

[13] Leon S. Searl, 'Space Based Internet Emulation Manager Design', September 28, 2001, pp 4-11.

[14] Leon S. Searl, 'Space Based Internet System Requirements', August 30, 2001, pp 4-5.

[15] Pooja J. Wagh, 'Design for a Satellite Communication Link in a Space Based Internet Emulation System', June 200l, pp 38-58.

[16] http://www.celestrak.com/columns/

[17] Sandhya Rallapalli & Sujit Baliga, 'An API for QoS in Linux', May 2001. http://www.ittc.ku.edu/~sandhya/courses/845proj.html

[18] Saravanan Radhakrishnan, 'Linux - Advanced Networking Overview Version 1', September 1999 http://qos.ittc.ukans.edu/howto/index.html

[19] http://ipinspace.gsfc.nasa.gov/faq.html

[20] http://ai3.asti.dost.gov.ph/sat/chara.html