The University of Kansas

Technical Report

Design for a Satellite Communication Link in a Space Based Internet Emulation System

Pooja J. Wagh
Gary Minden

ITTC-FY2003-24350-03

September 2002

Project Sponsor:
NASA

Glenn Research Center

Abstract

The Earth Observational Satellites (EOS) transmit their collected information to communication satellites, which relay this information to the ground stations. NASA employs *Tracking and Data Relay Services Satellites* (TDRSS) for communication from the EOS satellites to the Earth. But each EOS satellite is assigned a fixed time slot to access the TDRSS and to relay their data. So until their access time slot, the EOS satellites need to store the data on-board using high data rate and high capacity recorders.

The need for such huge storage components on the satellites could be eliminated if the satellites were capable of routing and switching data to other satellites and ground stations. The Space Based Internet (SBI) development project applies this concept to design and implement a prototype for achieving inter-networking between the satellites and ground stations.

To evaluate and test this prototype, the SBI project proposes to develop an emulation system that will model an actual satellite system. One of the major functions of this system would be to emulate the actual communication links between satellites and ground stations or between satellites. The satellite link provides a constant bit rate (CBR) service during the entire transmission duration but the signal transmission suffers from high propagation delays. This thesis describes the requirements and design for emulating an actual satellite transmission link, which would provide CBR control as well as introduce the propagation delays during transmission.

Table Of Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

1.1 Satellite Technology for Earth Observation:

The concept of satellite communication gained importance when the first artificial Earth orbiting satellite in 1957[4], Sputnik transmitted information from space to Earth. Since then, the satellites have been widely used for telecommunications and Earth Observation purposes. Earth Observing System satellites (EOS) have enhanced our understanding of the Earth and surrounding space through remote sensing and communication of those results back to the Earth.

In 1991, NASA launched a comprehensive program called the *Earth Science Enterprise (ESE)* [1] to study the Earth as an environmental system. By launching EOS satellites, NASA hoped to understand the impact of natural processes on the humans. The main aim of the ESE mission was to explore how the Earth's systems of air, land, water and life interact with each other and so this mission blended together fields like meteorology, oceanography, biology and atmospheric science.

The Earth Science Enterprise has three main components: a series of Earth-Observing Satellites, an advanced data system and a team of scientists to study the data. Phase I of this mission comprised of focused and free-flying satellites, Space Shuttle missions for airborne and ground-based studies. Phase II launched the first *Earth Observing System (EOS)* [2]satellites, Terra and Landsat-7. EOS supports a coordinated series of polar-orbiting and low-inclination satellites for long-term global observations. EOS satellites also coordinate their data collection with the other EOS satellites to provide information about a single event such as a *hurricane.*

1.2 Earth Observation Satellites

The Earth Observing Satellites focus on monitoring and predicting the future changes in the environment. The EOS satellites can make observations over a larger area than the terrestrial stations. EOS can observe and monitor places such as distant parts of oceans, deserts and Polar Regions, which cannot be accessed by terrestrial links and thus prove to be advantageous for earth observations.

Some of the important applications of the Earth observation satellites are:

- Remote sensing over land and water
- Atmospheric measurements
- Predicting natural disasters like floods, hurricanes
- Navigation
- Weather forecasts

1.2.1 Network Issues in Earth Observation Satellites :

Earth Observation satellites use number of technologies to collect observations and store raw data and communicating the information to the earth stations. These techniques include direct transmission, storage and deferred transmission and relay through communication satellites.

1.2.2 Communication from EOS to Earth stations

The communication in satellites is based on *Line of Sight principle.* The communication relay satellites receive and transfer the data information to the destination ground terminal when the ground terminal is in view of the satellite. The Earth Observation satellites use these relay communication satellites to transmit their information to the ground stations.

The Relay system currently employed by NASA EOS satellites is the *TDRSS* system. TDRSS stands for *Tracking and Data Relay Satellite* System [6]. There are currently

6 TDRSS communications relay satellites, which provide complete global coverage. These satellites are able to see the ground stations, which fall within its coverage area, at all times.

The EOS satellites take measurements over fixed points on the earth and then broadcast the data to TDRSS satellites. Through TDRSS, the EOS satellites can access the required ground station for 60 to 70% of its orbit time. Since there is a constellation of TDRSS satellites, the EOS satellites fall under the coverage area of any one TDRSS satellite or the other. TDRSS employs tracking services to locate the ground stations and relay services to transmit the information received from different sources to their respective destination ground stations. Fig1 shows the communication between the EOS satellites and the ground stations through TDRSS.

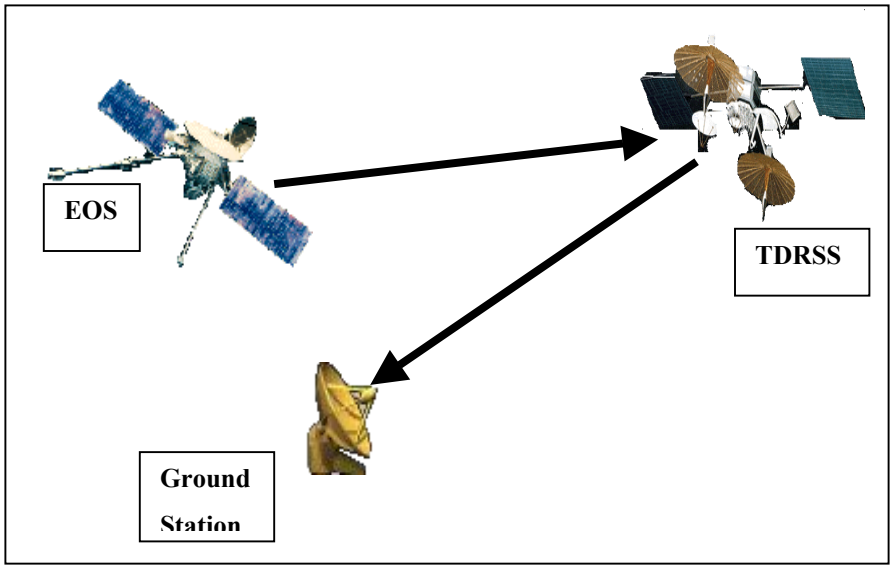


Fig 1: EOS to Ground Station Communication through TDRSS

1.3 Problem Definition

There are a large number of EOS satellites within the coverage area of a single TDRSS satellite and so each EOS satellite has a fixed time slot to transmit

information to TDRSS. Therefore, the EOS satellites need to have high data rate and high capacity recorders to store data on board. Also, the communications systems on the satellites need to be satellite specific, which means that each satellite should have its own communication frequency, protocol and command structure. This approach leads to incompatible and non-reusable communications components.

1.4 Proposed Solution

The Space Based Internet (SBI) development project proposes a solution to the current problems in satellite communication. It aims at establishing routing capabilities in the EOS satellites and ground stations. Such a mechanism will enable the EOS satellites to route data to the intended ground stations via other EOS satellites or ground stations and so eliminate the need for onboard data recorders.

The SBI project envisions that each EOS satellite participates in a Space-Based Internet. That is, each satellite in the SBI system would be capable of originating network traffic, terminating traffic and above all, switching traffic between other satellites and ground stations [3]. The satellites would carry a communications systems, which has several channels or beams (RF or optical) and the satellites would communicate with each other over Internet Protocol (IP). By centering the SBI system over a common communication protocol (IP) eliminates the need for having any satellite-specific communications systems or specialized ground station equipment [3].

1.4.1 SBI Approach

The SBI project proposes to design, develop and implement an initial prototype of the architecture for creating a Space Based Internet. The SBI network software implementing the routing and switching functionality will comprise of standard modules that can be deployed with minimal cost on the satellites and ground stations.

This approach will lead to a standard communications system on every satellite and ground station.

To evaluate the SBI system prototype, an SBI emulation system has to be developed. The emulation system will model the actual satellite system. Emulating a satellite system would involve execution of actual scenarios with the emulation nodes executing the SBI network software and applications programs.

The emulation approach reduces the complexities and the costs involved as against evaluating the SBI software on a real satellite system. The emulation setup consists of normal PCs acting as emulation nodes, which would run the SBI network software. Different scenarios can be executed and tested without having to change much of the setup. The emulation system has to emulate the communications systems hardware on the satellites, since it is not feasible to construct the satellite hardware on the SBI nodes.

1.4.2 Background for SBI Emulation System

The motivation and the foundation for designing the SBI system is derived from the work done by University of Kansas for the Defense Advanced Research Projects Agency (DARPA). The project Rapidly Deployable Radio Network (RDRN) developed network control programs and communications systems to demonstrate a rapidly deployable network

1.4.2.1 RDRN System

The RDRN system utilized wireless ATM technology to provide an adaptive and re-configurable network. It had interoperability with IP based networks and provided multi-hop operation over wireless nodes as far as 10 Km in distance.

Each RDRN node had RF communications systems and network control programs to establish the network topology and the optimum routes. In real environment, the nodes determined their location from Global Positioning Receivers. Knowing the locations of all the nodes, the system configured the network topology establishing point-to-point switched communication links. Further, the RDRN nodes calculated the optimum routes to the other nodes through the routing software.

The RDRN emulation system was designed to test the RDRN network for multiple scenarios. The emulation system helped to evaluate systems, which would be much larger than those possible through actual deployment.

1.4.2.2 RDRN Emulation System

RDRN emulation system was built to test and evaluate the RDRN software. It consisted of RDRN nodes and the Emulation Manager, which controlled the emulation and the communication network between the nodes.

The Emulation Manager was the bridge between the RDRN network nodes. In the emulation environment, the nodes initially registered themselves with the Emulation Manager and the Emulation Manager tracked the position of each RDRN node as per the scenario configuration file.

The network control software executed the topology algorithm to determine the network topology. Upon deciding the topology, the Emulation Manager established the point-to-point links via the underlying ATM network. The routing protocol configured the optimum routes for each node. The communications network was based on “virtual circuits” created on the underlying ATM network.

The SBI Emulation System can be designed from the RDRN Emulation System. The existing RDRN framework can be used with some modifications to the Communication Emulation System to emulate space communication.

1.4.2.3 SBI Emulation System.

The SBI Emulation System design extends the land-based RDRN emulation system to emulate space-based systems. The SBI emulation system models the entire satellite system and the emulation nodes represent either satellites or ground stations.

Following modifications can be done to the RDRN emulation system to build the SBI emulation system:

- The RDRN emulation node has to represent a ground station or a satellite in Earth's orbit.
- The land-oriented node location and topology algorithms have to be modified to incorporate mobile nodes in Earth orbit.
- The RDRN nodes communicate over short distances and at relatively low capacity. A SBI communication will be over much longer distances and will use high capacity links to handle observational data. Therefore, the communication emulation software for RDRN has to be modified to account of long-ranged, space-based communication systems.
- Communication Traffic models have to be developed according to the satellite systems and earth scientist's data gathering goals.
- Along with routing algorithms, the emulation system also has to include scheduling algorithms, which schedule the satellite instruments for data collection and gathering.

1.5 Scope of this thesis

This thesis describes the requirements and design for emulating the communication link between the satellites and ground station nodes in the SBI Emulation system.

Communication emulation involves modeling the communication channel and incorporating the features of the satellite transmission link for space-based communications. The following features will be emulated:

- The communication channel on the satellite, which forms the medium for wireless communication in space. Each satellite or a ground station has communication channel, which transmits the signals on the transmission link.
- Constant bit rate service to provide a guaranteed and fixed data rate on the link.
- Link Propagation delays during data transmission, which would be the actual path delays occurring on the satellite links.

1.5.1 Challenges

To emulate the communications systems requires consideration of the following aspects:

- A satellite or a ground station can form multiple links. Therefore each emulation node should facilitate multiple connections with other emulation nodes.
- The NASA EOS satellite orbits are somewhat irregular. So, even though it possible to predict, the communication resources at any point in time and space are varying. So it is highly challenging to provide a dedicated bandwidth for each instrument link on the satellite [3].
- The propagation delay in satellites is very high. It varies from 10-250ms one-way. To simulate such a high propagation delay on the emulation nodes would require a large number of packets to be queued at the transmission node.

1.5.2 Solution

This thesis provides the following solutions to the above mentioned challenges:

- A satellite establishes wireless connection with the receiver ground stations by pointing its instrument antenna in the direction of the receiver antenna. A satellite might have multiple instruments facilitating multiple connections. In the SBI Emulation network, each satellite link is emulated as an IP over Ethernet

connection between two emulation test nodes. To model the multiple instrument-links would require an equal number of physical Ethernet interfaces on a test node. The first section of the thesis deals with creating virtual Ethernet interfaces on a single physical Ethernet device to enable multiple connections on a test node. The virtual Ethernet devices will model the behavior of multiple instrument communication channels on the satellites.

- Each satellite link has a fixed dedicated bandwidth. The link capacity is specified by the instrument data rates. The total capacity on the satellite is the sum of all the link capacities. In the emulation environment, each node will have bandwidth equal to the satellite capacity. The node shall reserve the bandwidth on the different links as per the satellite specifications. The bandwidth on each link should be fixed and guaranteed, as each link would be dedicated to either routing or data collection. The second part of the thesis describes the design for providing a constant bit rate on the link by utilizing traffic control Quality of Service algorithms in Linux.

- The satellite links establish communication over long distances. The signal transmission on these links suffers from high propagation delay. The propagation delay ranges from 50 to 250ms depending on the distance between the satellites and the ground stations or other satellites. The propagation delay on a particular link is also not constant as the satellites are in constant motion. The third part of the thesis describes the mechanism for simulating the propagation delay during data transmission on the emulation test node.

1.6 Organization of thesis

The rest of the thesis is organized in the following manner. Chapter 2 describes the background theory for this thesis, the fundamental principles of satellite communication and gives an overview of the SBI system emulation architecture.

Chapter 3 describes the design for creating the Virtual Ethernet devices on the emulation nodes. The Virtual Ethernet devices will model the behavior of the communication channels on the satellite or ground station. Chapter 4 talks about providing Constant Bit Rate Control on the emulation link and Chapter 5 describes the mechanism for simulating the path delay on the link. The final chapter states the conclusions and the scope for future work.

Chapter 2

2 Background Theory

The communication systems on the satellites consist of several channels (RF or optical). Each channel establishes a transmission link for communication. This chapter gives a brief overview of the current satellite systems and then proceeds to describe the SBI emulation system that is designed to emulate the satellite system.

2.1 Satellite System : Overview

This section gives an overview of the current satellite system, its components and the characteristics of the transmission link. The satellite system comprises of ground segment and the space segment. The space segment constitutes the satellites while the ground segment is the earth ground stations.

2.1.1 Space Segment

A satellite establishes a line-of-sight wireless link with the ground stations. Since the transmission and the reception frequencies for the satellites are different, the satellite has to communicate with two types of earth stations. The *uplink* earth station modulates the signals and radiates it to the satellite, which in most cases is a relay satellite[5]. The satellite receives the signal and shifts the signal frequency, amplifies it and then re-radiates it back to the earth where it is received by *downlink* earth stations [5]. The EOS satellites rely on the TDRSS system to relay their information.

Satellites communicate with the ground stations or other satellites through a transmission link between the source and the destination antennas. Antennas on the satellites and the ground stations provide directionality and focus for the signals to be transmitted. In absence of the antenna, the signals would radiate in all directions in space and the quality of the signal reaching the destination would be low.

Every satellite has a different mission and so each satellite is designed individually, that is each satellite has its own communications frequency, protocol and command structure.

2.1.2 Ground Segment

The ground segment comprises of the uplink and the downlink earth station components. They include:

- Multiple beam antennas for simultaneous communications with other satellites.
- Precision Systems for tracking satellites.
- Uplink and downlink Communication Equipment.
- *Telemetry Tracking and Command (TT&C)* systems for monitoring the performance of the satellites and receiving telemetry data from the satellites.

2.1.3 Satellite Transmission Link

The transmitter antenna on a satellite establishes a communication link with the receiver antennas for relaying information. The following factors are important for evaluating the link performance:

- The ability of the link to provide a guaranteed Constant bit rate service.
- Link Propagation Delay.
- Satellite Channel Bit Error Rate (BER)

2.1.3.1 Transmission Capacity

The transmission capacity depends upon the allocated satellite bandwidth for the link. Each communication channel on the satellite has reserved bandwidth to provide Constant Bit Rate Service for voice, video and data traffic. The satellite bandwidth has to be efficiently utilized in case of multiple links.

2.1.3.2 Transmission Delay

Due to large distances between the satellite and the Earth, the signal, travelling at the speed of light takes a long time to propagate to the earth and then back to the satellite. A complete round trip propagation delay for a satellite link between a GEO satellite and the earth, which is approximately 36,000, Kms is around 250ms. This large delay has an adverse effect on the transmission of voice and video traffic. In case of data communications involving transmission speeds of 10Mbps and higher, huge amount of data transmitted by a source is temporarily in flight on the satellite link due to such a high propagation delay [4].

2.1.3.3 Satellite Channel Bit Error Rate(BER)

BER is defined as the number of transmitted bits received with errors. It is expressed as a proportion of the total number of bits transmitted. It is specified in the following form: N in 1 x 10^x where N is commonly unity.

The channel bit rate is a function of the weather conditions along the propagation path. It is unpredictable and variable and during heavy rain storms or cloud cover BER can be higher than 1 in 10^6. Larger distances between the satellites and the earth stations can result in a higher BER. BER as low as 1 in 10^{10} can be achieved by employing Forward Error Correction (FEC) technique [4].

2.2 SBI System Architecture

The previous section described the elements of the current satellite system. This section details the architecture of the SBI system The SBI project proposes to establish internetworking through wireless Ethernet technology between Earth Observation Satellites (EOS).

The SBI system is designed to facilitate IP over Ethernet connections between the satellites and ground stations or other satellites. Since the communication is based on

a common protocol, it eliminates the need for designing any satellite-specific communications systems or any specialized ground station equipment. The SBI system software modules also can be deployed on the satellites and ground stations with minimal cost.

The proposed SBI system design is divided into 2 parts:

- SBI Emulation System:

  This system emulates the satellites and ground stations on an emulation testbed. The components of the emulation system model the real hardware and the actual communications systems through software. It also emulates the satellite communication links between the emulation nodes. The SBI emulation system is designed solely for emulation purposes and is not a part of the SBI system to be placed on actual satellites and ground stations [7].

- SBI Node Network:

  The SBI network software creates the space-based Internet between the actual satellites and ground stations. This architecture employs software modules that can be loaded on to the actual satellites and ground stations to enable them to switch network traffic. The SBI network software is evaluated on the SBI emulation system.

2.2.1 Features of the SBI system

2.2.1.1 Types of Satellites

In the emulation environment, each SBI node represents a satellite or a ground station. There are 3 types of nodes:

- Data Source and Relay Satellites
- Relay Satellites
- Facilities or ground stations

Data Source and Relay Satellites:

These satellites are mostly EOS satellites. These satellites are responsible for collection of data and also relaying the information to the other SBI nodes in the emulation. For SBI satellites that are under consideration, the instrument rates might range from a few bits per second to at least 150Mbps [7]. The bit rate on the communication link should be able to handle the peak data rate of all instruments that can make observations simultaneously.

A Data Source and a Relay satellite should also have at least two transmitters and receivers to establish data links for routing data. The data links will have a dedicated bandwidth which should be sufficient to carry the peak rate of all the satellite instruments and plus the data that is routed from the other satellites.

In the actual satellite environment, these satellites would represent Low Earth Orbiting (LEO) satellites. LEO satellites orbit at an altitude less than 2000 Kms from the earth's surface.

High Capacity Relay Satellites:

These satellites are used solely for relay purposes. They act as router satellites switching traffic between other satellites and ground stations. These satellites have high data rates and need at least 2 dedicated links to function as a router satellite.

A high capacity relay satellite represents satellites in the Geo-stationary (GEO) or Medium Earth (MEO) orbits. GEO satellites are placed at 36,000 Kms above the earth's surface. These satellites have a larger coverage area of the earth and therefore are able to see most of the LEO satellites and ground stations within their coverage area. These satellites can function as good routers. MEO satellites are about 10,000 Kms above the earth's surface and have less coverage area than GEO satellites.

Facilities:

A facility represents a ground station on the earth's surface. The SBI node representing ground stations should have at least one antenna. It can form link with any type of SBI satellite.

2.2.2 SBI Networking

The SBI networking models the satellite transmission links and emulates the actual communication between the satellites and the ground stations. The data communication between SBI emulation nodes is IP-based networking over Ethernet. The communication links established between nodes have two types of data rates:

- Low Data rates for Data Source and Relay Satellites for data collection
- High Data Rates for the Relay Satellites and the Facilities for routing purposes.

The SBI communication is based on a common Internet Protocol (IP). This eliminates the need for satellite-specific communications systems and specialized ground station equipment in the real satellite systems. IP based communication will also allow the SBI system to evolve with technological advances.

The SBI nodes contain software modules that will use adaptive algorithms to figure out the network topology and route the network data through optimum routes.

2.2.3 SBI Software

There are three types of nodes in the SBI emulation environment:

- Emulation Manager – controls and monitors the entire emulation. This node is a part of the Emulation System.
- Central Operations Node – acts as the *TT&C* Earth Station for configuring the network topology and figuring the optimum routes for the entire network.
- SBI Nodes – represent the satellites and ground stations.

The SBI software is in the form of modules that can be loaded on to the emulation nodes. The software can be classified into two categories:

- SBI Emulation Software
- SBI Node Software

SBI Emulation Software:

SBI Emulation Software emulates those portions of the satellite and the ground station communications hardware, which cannot be constructed on the SBI emulation nodes[7]. It emulates the communication channels on the satellites and ground stations and models the communication links between the channels.

SBI Node Software:

SBI Node Software refers to the software modules that would be placed on the satellites and the ground stations in an actual satellite system [7]. This software is resident on all the SBI nodes but mainly on the Central Operations Node. The Central Operations Node software contains modules, which implement adaptive algorithms for configuring the entire network topology. It figures out the optimum connections between the nodes based on the scenario parameters obtained from the emulation manager. It decides the routing tables for SBI nodes and communicates with the other nodes to transmit all the routing information.

The Node Software on the other SBI nodes is responsible for receiving the routing information from the operations node. Based on the information obtained from the operations node, the SBI nodes initiate request for creating or deleting inter-nodal connections. This software also relays satellite-instrument data through the established connection links. The inter-nodal connections emulate the actual communication links in the satellite system.

2.2.4 SBI Emulation System Hardware

The SBI emulation system hardware contains two networks involving the Emulation Manager. The SBI nodes form a Data Network based on Ethernet connections through a managed Ethernet switch. The Emulation Manager configures the connections between the SBI nodes. The second network runs through an unmanaged Ethernet switch and is used by the Emulation Manager to send control commands to the other nodes and also sense the status of the emulation. The SBI nodes receive their configuration parameters from the Emulation Manager through the second network.

Figure 2 illustrates the SBI System Architecture. The Emulation Manager interacts with the other SBI nodes on both the networks. The details of each entity will be explained in details in the later sections.

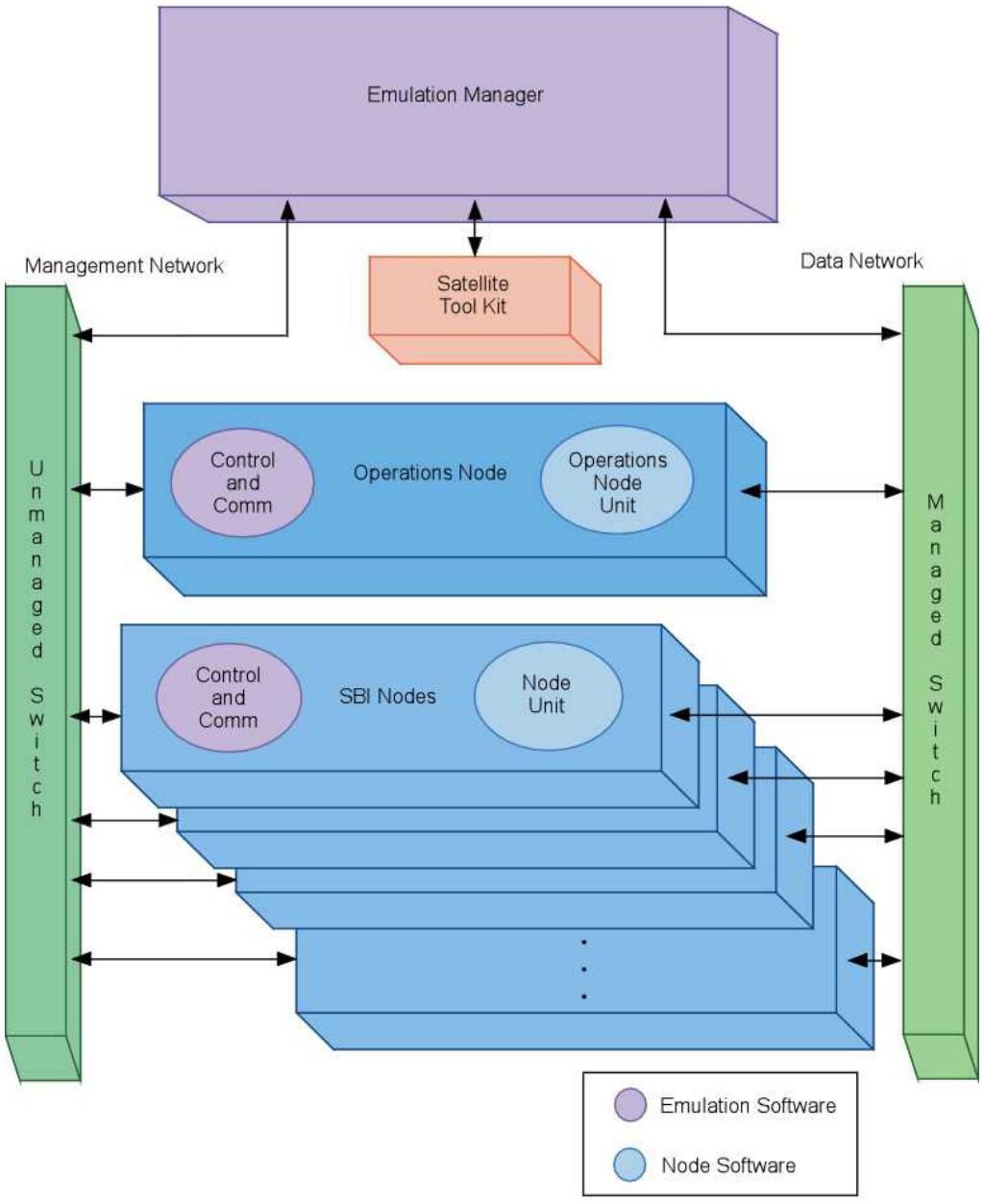


Fig 2: Space Based Internet System Architecture

2.2.5 SBI Emulation System

This section describes the emulation system architecture for the SBI environment. The emulation system tests and evaluates the SBI network software. A SBI node in

the emulation system represents a satellite or a ground station as specified in the emulation scenario.

The components of the emulation system software are:

- Emulation Manager, which is responsible for control and administration of the entire emulation scenario.
- Node Emulation Software on the other SBI nodes. This software contains modules to emulate the satellite communication channels and transmission links and modules to interface with the actual Node software.

2.2.5.1 Emulation Manager

The Emulation Manager is the central controlling entity for the emulation system. It is a user-level application and hosts the configuration files necessary for creating and executing the emulation scenario [8]. It is responsible for emulating the communication links on all the other SBI nodes.

The components of the Emulation Manager are:

- Communications Controller – controlling the communication emulation.
- Event Controller – responsible for event scheduling.
- Orbital Manager – performs orbital calculations and computations for the emulation scenario.
- User Interface Manager – Interfaces with the user for input for the emulation scenario.

The various components of the Emulation Manager are described below. Figure 3 shows the Architecture for the Emulation Manager and its modules.

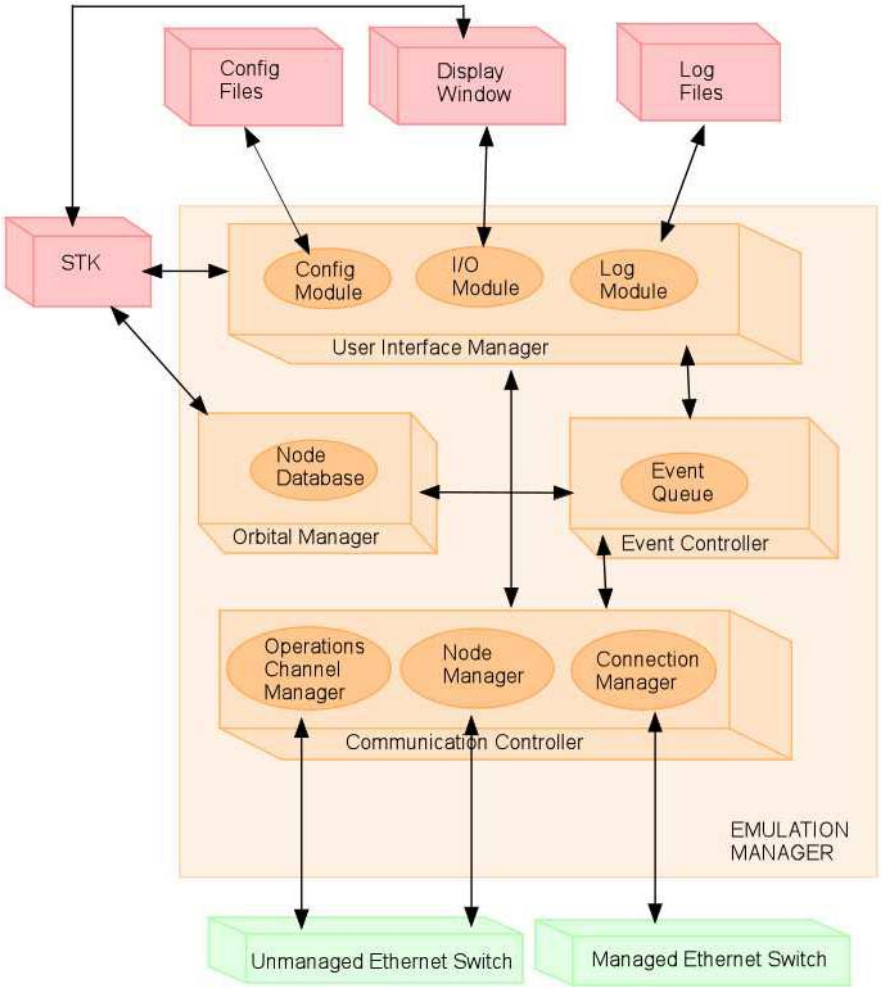


Fig 3: SBI Emulation Manager Architecture

2.2.5.1.1 Communication Controller

The communications controller controls the configuration and control of the emulation nodes. It hosts three types of manager modules to communicate with the modules on the other SBI nodes.

- Node Manager

The Node Manager is the interface to the other SBI nodes. The control and the configuration commands from the Emulation Manager are transmitted to the other SBI nodes through this manager module.

- Connection Manager

The connection manager controls the connections between the SBI nodes. These Ethernet connections emulate the communication on the transmission links in satellite system. The connection manager receives requests from the SBI nodes to create or remove the connections. The connection manager accordingly creates or removes connections on the managed Ethernet switch.

- Operations Channel Manager

The Emulation Manager acts as a medium of communication between the Central Operations Node and the other SBI nodes. The Operations Node represents the TT &C earth station, which sends commands to the other satellites and ground stations. In the real world, the TT&C earth stations use S-Band communication for these purposes but in the emulation environment, the commands from the Operations Node go through the emulation manager to the respective nodes [8].

The operations channel manager receives the commands from the Central Operations Node and forwards them to the respective nodes. The commands are mostly instruments scheduling, data transfer scheduling and those related to routing information.

2.2.5.1.2 Event Controller

Event controller schedules the events on the Emulation Manager. This simplifies the task of the emulation manager in controlling the entire emulation. Emulating a satellite scenario having many satellites and ground stations involves a lot of orbital

calculations for the emulation manager. The emulation manager also, has to configure each node according to the specifications. It has to keep track of connections between the nodes and also issue control commands to the nodes. All these actions are considered as *events*.

The event controller maintains a list of events that are set to occur according to the time of occurrence in an *event queue.* The *event manager* module is responsible for the actual scheduling of the events.

2.2.5.1.3 Orbital Manager

The orbital manager is responsible for all the orbital calculations pertaining to the emulation scenario. The orbital calculations comprise of the orbital positions and propagation calculations. It also houses a *Node Database,* which keeps information regarding each emulation node. The information contains the node name, type of orbit, mission, positional and vehicular data and data link rates [8].

The orbital manager interfaces with the *Satellite Tool Kit (STK)* [17] from Analytical Graphics, Inc. to get the node information and performing orbital calculations. The node database information is conveyed to the Central Operations Node for deciding on the network topology and the optimum connections.

2.2.5.1.4 User Interface Manager

The user interface manager interacts with the user for purposes of executing the scenario. It has an *I/O* M*odule* to control the user interface displays. The user can issue commands to start, stop, pause and resume requests to the scenario. The *Config Module* retrieves data from the configuration files regarding all the attributes of the scenario to be executed and routes it to the Orbital Manager. The *Log Module* writes out log data periodically to the log files corresponding to events occurring in the emulation.

2.2.5.2 Node Emulation Software

The Node Emulation Software resides on the other nodes. This unit interacts with the components of the SBI network software and implements the communication emulation unit on the nodes. Figure 4 describes the modules of the Node Emulation Software.

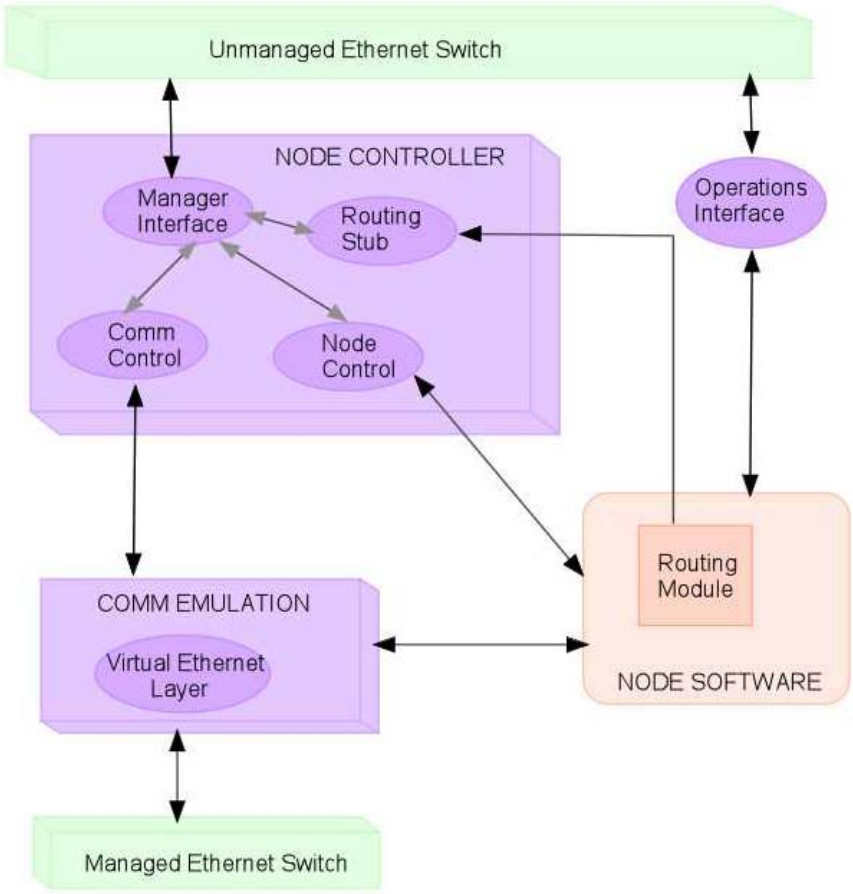


Fig 4: SBI Node Emulation Software Modules

The following sections describe the Node Emulation Software.

2.2.5.2.1 Node Controller

The node controller provides interface to the managerial modules on the Emulation Manager and also interacts with the Node software and the communication emulation unit on the nodes. It has the following modules:

The *Node Control* module receives commands from the Emulation Manager and takes appropriate actions. The actions include elaborate requests such as stop, start or pause or involve forwarding of commands to the Node software [8].

The *Manager Interface* receives orbital data for the node from the EM and routes it to the Node software. On the Central Operations Node, this module receives orbital data pertaining to the entire scenario from the Emulation Manager and sends it to the Node software for orbital calculations.

The *Communications Controller* module controls the behavior of the communication emulation unit. The communication emulation unit is responsible for emulating the satellite communication between the nodes. This module implements multiple communication channels by using virtual Ethernet devices for communication. This module also retrieves from the Emulation Manager, the values for data link rates and propagation delay on the communication channels, and sends it to the communication emulation unit.

The *Routing Stub* module conveys the Emulation Manager in case of change in any routing connections. The Emulation Manager acts on this data by adding or removing a connection on the managed Ethernet switch.

The *Operations Interface* on the Operations node transmits scheduling commands and routing information to the Emulation Manager. The other SBI nodes receive the information through the same interface.

2.2.5.2.2 Communication Emulation Unit

This unit is responsible for inter-nodal communication. This unit emulates the characteristics of an actual communication link. The satellite instrument data generated by the emulation nodes is sent through this unit to the connected nodes on the managed Ethernet switch. It models the following characteristics of satellite communication:

- Communication channel – The communication emulation unit models a communication channel for a satellite instrument by creating a virtual Ethernet device on the emulation node. A satellite communication link is established between the virtual devices on the emulation nodes. To facilitate multiple connections on the nodes, multiple virtual devices have to be created.
- Communication link – The transmission link features are modeled on the Ethernet connection between two emulation nodes. It provides the following features on the link :
    - Constant bit rate service by reserving bandwidth on each link through Quality of Service.
    - Link Propagation delay to simulate the actual link transmission.

The nodes create and configure the virtual devices through the C*omm Control* module.

2.2.6 SBI Node Software

The SBI node software is the actual software that is to be deployed on the satellites and ground stations in a real environment. In the emulation world, this software resides on the nodes other than the Emulation Manager. This section gives a brief overview of the network software and its interaction with the communication emulation unit, which is the main focus of this thesis.

The network software can be divided into 2 major sections:

- Operations Node software – This piece of software resides on the Central Operations Node, which emulates the TT&C earth stations.
- Node Program Software – This software module resides on all the nodes, other than the emulation manager

2.2.6.1 Operations Node Software

This module is responsible for all the connections between the SBI nodes. It determines the network topology and configures all the routing tables for all the SBI nodes. Figure 5 shows the different modules of the operations node software.

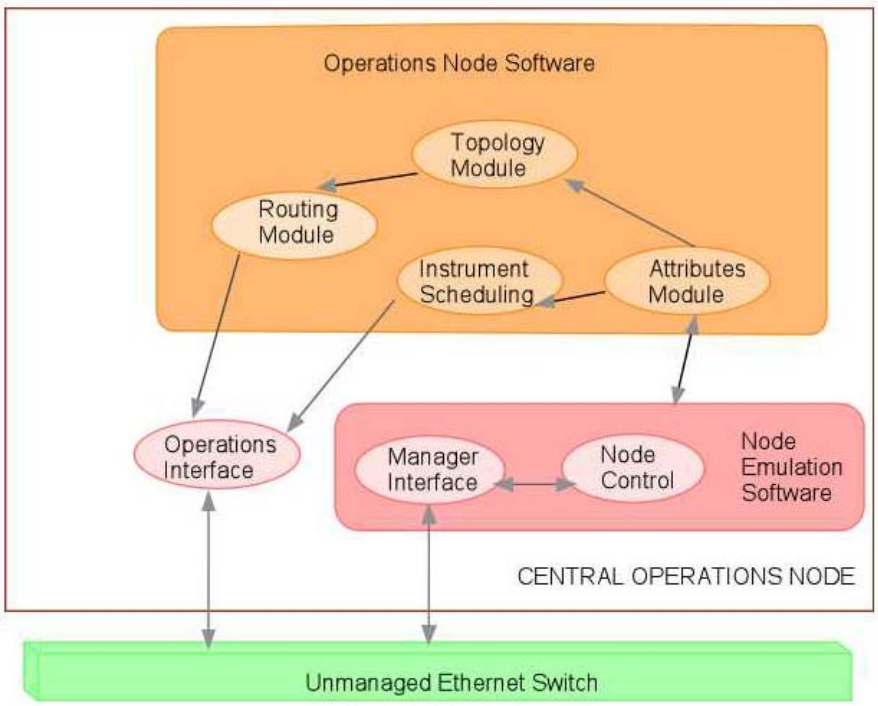


Fig 5: SBI Operations Node Software Modules

The *Attributes Module* receives node configuration parameters through the Node Emulation Software and passes it on to the *Topology Module.* The Topology Module

determines the network topology and the optimum connections between the nodes. The *Routing Module* configures the routing tables for all the nodes by considering the optimum routes to the destination. The *Instrument Scheduling* module receives the attributes from the *Attributes Module* and is responsible for scheduling the satellite instruments on the nodes. The routing and the scheduling information is transmitted to the respective nodes via the Emulation Manager. The Operations Node utilizes the Operations interface for this purpose.

There are various factors on which the topology algorithm decides on the connections:

- Occlusion by Earth i.e. Loss of *Line of Sight (LOS).* The satellites communicate on the LOS principle. Therefore the decision for connections can be made on whether the satellites and the ground stations can see each other or not.

- Instrument-link capacity. Each connection requires a dedicated bandwidth. In case of multiple connections on one node, the bandwidth of the node has to be utilized efficiently so that multiple dedicated links are possible. The topology module decides any new connection only after considering the total bandwidth utilization

- Duration of Line of Sight between two nodes. The nodes establish connection only if they have Line of Sight with each other as per the emulation scenario. The duration of LOS is an important factor in deciding the connections. A longer duration indicates a longer dedicated connection, which allows the satellites to route more data on the connection.

The connection decisions for the scenario are based on the routing tables and are conveyed to the Emulation Manager. The emulation manager conveys the routing information to the rest of the nodes.

2.2.6.2 Node Program Software

Figure 6 shows the details of the Node Program Software.

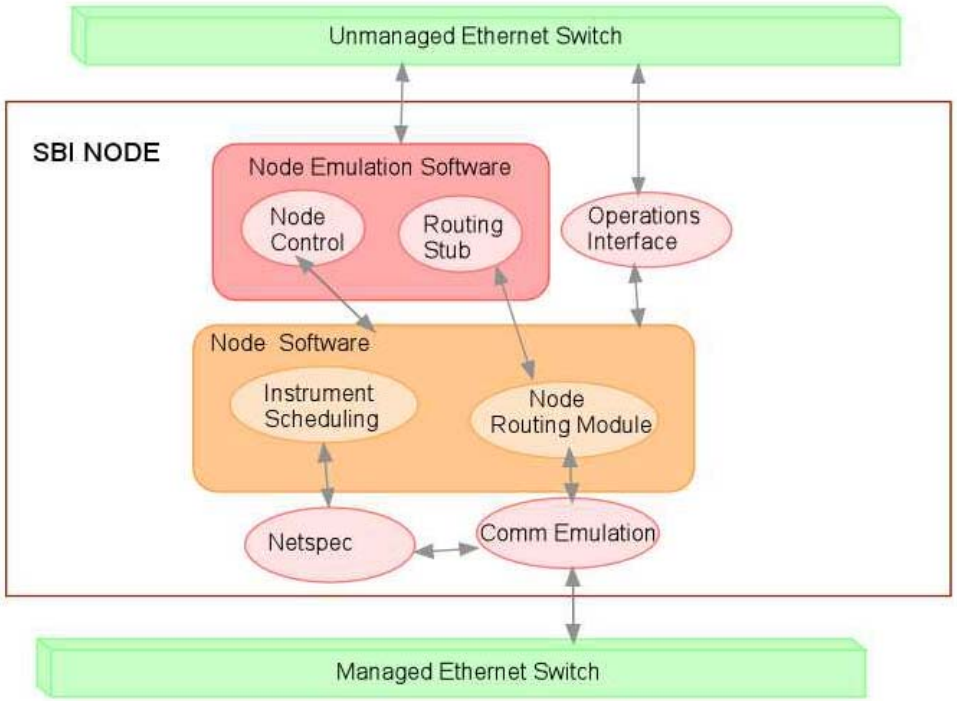


Fig 6: SBI Node Software Modules

The SBI nodes receive the routing and scheduling information from the Emulation Manager to the Node Software module. On the basis of routing information, it decides on the connections and then conveys to the Emulation Manager to actually create or remove the connections on the managed Ethernet switch.

The *Instrument Scheduling* module receives the instrument-scheduling commands from the Emulation Manager. The actual satellite instrument data is emulated on the Ethernet connection between the nodes. The scheduling commands for the nodes prompt the node- instruments to turn ON and enable the transmission of the instrument data through the virtual Ethernet connections. The instruments on the relay

satellite (which are virtual Ethernet connections in the emulation world) have to remain activated continuously since they relay data from different satellites or ground stations. Incase of Data Source and Relay satellites, some of the instruments are dedicated to data collection. These instruments collect data periodically and so they need to be turned ON only at the time of data collection.

In the emulation system, instrument data is modeled as NetSpec [9]scripts. NetSpec emulates different types of satellite traffic through the scripts and runs those scripts as per the configuration of the nodes. So NetSpec data traffic along the communication links emulates actual satellite transmission in the emulation system.

Chapter 3

3 Virtual Ethernet

The previous chapter provided the background necessary for understanding the details of communication emulation between the nodes in the SBI system. The communication in an actual satellite system is established on a transmission link between the transmitter and the receiver antennas. Each antenna beam can be divided into different channels facilitating multiple connections to the satellites. In the SBI emulation system, the transmission link is emulated as an IP over Ethernet connection between two emulation nodes. In order to have multiple connections from a single node, each node would require multiple Ethernet interfaces. To eliminate the need for several physical Ethernet interfaces, *virtual Ethernet (VETH)* devices can be created on a single physical Ethernet device.

This chapter initially lists the requirements for the implementation of the VETH devices and explains the user-level control program written to create and configure the devices. The later sections of the chapter cover the implementation details of Virtual Ethernet.

3.1 Requirements for VETH Layer

Virtual Ethernet devices are Linux kernel level abstraction for the operating system, which considers these devices as actual network devices[10]. Each virtual Ethernet device emulates a satellite communication channel. The communication link is emulated as a connection between two virtual devices.

- Virtual devices implement the same functionality as the physical Ethernet devices. These devices are created through software code and should be transparent to the other layers of the Linux kernel. The virtual devices are implemented on the top of physical Ethernet but they do not perform any

processing on the data traffic. They create a virtual Ethernet layer between the network layer (IP layer) and the physical Ethernet layer.

- Each virtual device has an IP address and a unique 6-byte *Medium Access Control (MAC)* address, which is different from the underlying physical device. The first 3 bytes represent the vendor portion, which is the ITTC vendor ID. The remaining 3 bytes of the MAC address are unique for each virtual device. Figure 7 represents the MAC address representation of the virtual devices.

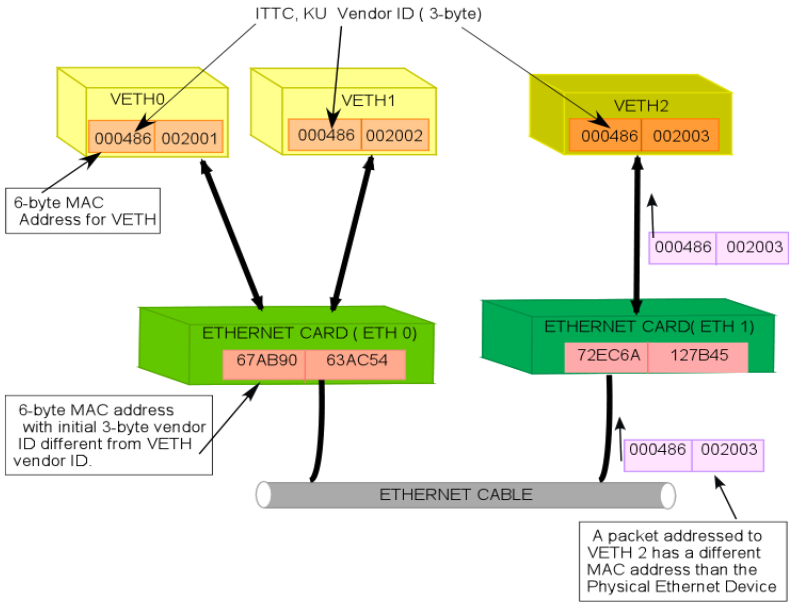


Fig 7: Mac Address Representation for the Virtual Devices

The ITTC vendor ID is 00:04:86. The idea of having our own vendor ID was that the packets meant for SBI network would have Source and the destination MAC addresses having the ITTC Vendor ID. This would help in differentiating packets that are not meant for SBI network. The traffic meant for SBI network would pass through the VETH layer before going to the IP layer, while the other IP traffic would by-pass the VETH layer.

- **Network Communication through the SBI network:**

  Transmission of packets from the VETH device

  The SBI network has the Virtual Ethernet (VETH) layer between the IP and the physical Ethernet layer. The network traffic from the IP layer is transmitted to the VETH devices before transmission to the physical device layer. The virtual devices provide Constant Bit Rate Service to the transmitted packets and introduce propagation delay during packet transmission to the physical layer.

  Reception of packets on the VETH device

  The VETH device receives the data packets sent above by the physical Ethernet device. The packets meant for the SBI network are de-multiplexed to the right virtual device on the basis of the destination MAC address. The packets not generated by the SBI network should have the MAC address of the physical Ethernet device. The virtual devices send the packets to the IP layer.

- These virtual devices can be created and configured through a user-level control program by *ioctl( ) system* calls to the Linux Kernel. *Ioctl( ) calls* allow the user to access the kernel implementations and insert the Virtual Ethernet layer in between the IP and the physical layer without modifying the existing Kernel structure.

3.2 VETH Architecture

VETH devices appear to the higher layers of the Linux kernel as hardware devices. In reality, these devices are created and configured by a software code at the user-level. The method by which the devices insert themselves between the IP and the Physical device layer is simple. For the Linux Kernel, the device is constructed as a C structure. The fields of *struct device* store information relevant to the VETH device and the function pointers can be set to point to the appropriate functions for device operations.

To create the VETH device, the user-level *ioctl ()* call creates an instance of the *struct device* C structure and registers the device with the kernel. This is similar to how physical devices are registered on kernel boot up. During kernel boot, the device drivers (for the physical devices) probe the PCI (or ISA) bus for devices [10]. Once the devices are found, they allocate and initialize the fields of the *struct device* structure for these devices and assign the appropriate function pointers. These devices are then registered with the appropriate kernel entity.

The insertion of the Virtual Ethernet layer shouldn't affect the operation of the physical network hardware. This layer is transparent to the traffic not meant for the SBI network. The *struct device* structure has function pointers for device operations such as *open, close, send, receive* etc. Since each of the virtual devices have the *struct device* structure, these function pointers can be set to the appropriate functions in the VETH layer without affecting the existing framework of the Linux kernel.

The Linux Kernel also provides ability to divert the SBI traffic coming from the IP layer during transmission or from the physical Ethernet layer during reception to the Virtual devices. The transmitted packets from the IP layer do not have to be multiplexed as the packets contain the name of the device on which to transmit. But the SBI packets that are received on the physical layer have to be routed to the correct

virtual device before the IP layer, which requires a multiplexing and de-multiplexing mechanism at the physical layer.

The virtual Ethernet devices implement the functionality of the physical Ethernet devices and so these devices appear as physical devices to the higher layers.

The next section talks about the user-level control program before describing the implementation details. The control program is responsible for creating and configuring the virtual devices.

3.3 VETH Control Program

The user-level control program creates and deletes the virtual Ethernet devices. The control program allows per-instance user-level configuration of these devices and implement the Ethernet functionality in them. Since the VETH devices exhibit the same properties as physical Ethernet devices, they can be also configured by *ifconfig*.

Fig 8 explains the layers for the SBI network and the interface between the user and the Linux kernel for implementing the Virtual Ethernet Layer. The VETH layer is a insertion between the IP and the physical layer. The network traffic generated from the SBI nodes passes through the VETH layer. Since the communication protocol is IP, the virtual devices receive the traffic from the IP layer. The traffic that is not meant for the SBI network does not go through the Virtual devices.

There are two types of user controls for the VETH layer:

- The control program *vethctl* – to create and destroy the devices.
- *ifconfig* - To configure the devices.

The control program configures the virtual devices while the *ifconfig* command can be used to set the network properties for the device.

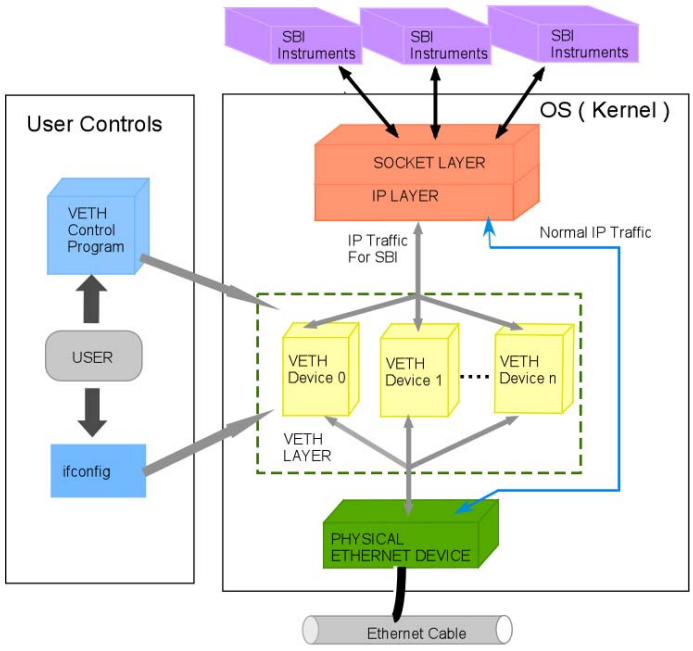


Fig 8: SBI Network Layers and Controls

3.3.1 Ioctl ( ) system call

The control program is responsible for creating the Virtual Ethernet Layer using the *ioctl ( )* system call. The *ioctl call* allows per-instance configuration of each device and registers the device with the kernel [10]. The control program creates an *INET Socket* for the *ioctl* system calls.

```
if( (ioctl_fd = socket(PF_INET, SOCK_DGRAM, 0)) < 0 ){
      fprintf(stderr, "\n Error in creating a Unix socket %s
\n",  strerror(errno));
      exit(1);
    }
```

The control program provides three options to control the devices. The 3 options are:

- Create Option

- Delete Option
- List Option

These options are command line based and perform operations as specified by the command option.

3.3.2 Create Option

The '*create'* option creates an instance of the virtual device. The virtual devices created are numbered sequentially and are named as *vethN,* where N stands for the interface number.

The syntax for creating the VETH devices is:

```
Vethctl –c <physical device name> <source MAC address>
          Where,
          –c: create option
          Physical device name: Name of the physical Ethernet
                                Device Ex: eth0
          Source MAC address:   6-byte MAC address for the
                                Virtual device
                                Ex: 00:04:86:00:00:01
```

This option initiates an ioctl( ) call to the Linux kernel relating to this operation. The appropriate function at the VETH layer is called to create the devices. The interface number assigned to the device is incremented, starting from '0' as the devices are created. An example for creating a virtual device is provided in APPENDIX A.

Some important conditions to be checked while creating the devices:

- The validity of the underlying physical device should be checked.
- The MAC address at the command line should have the first 3 bytes (vendor ID) equal to the ITTC vendor ID. (00:04:86).

The '*create'* option returns the name of the interface created. Ex: *veth0.*

3.3.3 Destroy Option

The '*destroy'* option deletes the virtual interface specified by the argument. The virtual device is a piece of software inserted in the Linux kernel. So when the virtual device is destroyed, the memory occupied by the device has to be released.

The syntax for destroying a VETH device is:

```
Vethctl –d <VETH interface number>
          Where,
          –d: destroy option
          VETH Interface Number: The interface number of the
                                 Virtual Device to be
                                 deleted.
```

The validity of the virtual device is checked before deleting the interface. An Example is provided in APPENDIX A.

3.3.4 List Option

The '*list'* option provides information on the virtual devices that are created. The information includes the names of the virtual devices and the underlying physical devices. An example is provided in APPENDIX A.

The syntax for the '*list*' option is :

```
Vethctl –l
          Where,
          –l: list option
```

3.3.5 Ifconfig commands

'ifconfig' command is used for configuring the virtual devices. Some of the important device properties configured through '*ifconfig*':

- IP address of the VETH device
- Netmask for the VETH device
- Size of Maximum Transmission Unit (MTU)
- Setting the device flags for multicast, promiscuity, broadcast etc.

The details of the implementation of the VETH layer are covered in the next section

3.4 Implementation of the VETH layer

Virtual devices are implemented in the Linux kernel as pseudo device drivers. Through this implementation, with a single physical network device, the system can emulate network traffic on an unlimited number of virtual network devices. These devices have to be created on top of the physical network device. The current implementation supports virtual Ethernet devices over physical Ethernet.

3.4.1 VETH Top Level Design

The main function of the VETH layer is to facilitate multiple connections on a single physical Ethernet device. Each virtual device has a C structure associated with it, which stores information relating to the virtual device. Program 3.1 refers to the *veth_device* structure, which is instantiated for each device. It contains information about the device name, interface number, the source MAC address and information about the physical network device. It also contains *struct device,* which is the kernel structure for representing a network device and whose function pointers can be set according to the requirements of the VETH devices.

When the virtual device is created, the contents of this structure are filled as well as the *device* structure pointed by *vethDev.* The *veth_stats* keep statistics of the packets flow through the VETH device. The information about the underlying physical device is stored in a structure *phy_device.* Program 3.2 explains the details of both the structures.

Program 3.1:The Virtual Ethernet Device structure

```
struct veth_device {

      char name[5];               /* Name of the device (veth0)
      char phyDevName[5];         /* Name of the physical device
      int  itfNum;                /* Interface Number

      struct device *vethDev;     /* Device structure associated
                                  /* with the virtual device
                                  /* Ex - veth0
      char srcMac[6];             /* Source Mac address
      struct veth_stats *nwStats;/* VETH device statistics

      unsigned long  vdevCreate  ; /* this flag is set when the
                                     /* Virtual devices created.

      struct veth_device  *next;

}
```

Program 3.2: The veth_stats and phy_device structures for the VETH devices

```
struct veth_stats {          /* statistics for the VETH device
      int tx_sent;
      int tx_dropped;
      int rx_sent;
      int rx_dropped;
}

struct phy_device {

      char phyDevName[5];    /* name of the physical device
                             /* Ex -eth0
      int num_vethdev;       /* Number of virtual devices
                             /* on the physical device.

      int (*change_mtu)(struct device *dev, int new_mtu);
                             /* A pointer to store dev-
                             >change_mtu /* pointer for the
                             physical device

      int (*pdev_recv) (struct sk_buff* , struct device* ,
                               struct packet_type*);
                             /* Function pointer pointing to
                             /* 'Receive' function of the IP
                             layer

      struct phy_device *next;

}
```

The packets coming on the physical Ethernet device have to go through the virtual device before going to the IP layer. The packets queued in the net bottom half after coming on the hardware device are checked for their packet_type fields. If the packet_type is IP then the 'receive' function for the IP layer is called. But the VETH devices intercept the packets going to the IP layer by changing the pointer of the 'receive' function for the IP packet_type. The *func* pointer, which contains the *ip_rcv* function for the IP packet_type is changed to the 'receive' function of the VETH device. The original function is stored in the '*pdev_recv'* function pointer. The

original pointer is restored back when the virtual devices on the underlying physical device are deleted. Similarly, the '*change_mtu'* function stores the function pointer for *dev->change_mtu* for a physical device. This is explained in Program 3.2.

The virtual devices provide CBR service to the packets and also introduce propagation delay in the '*send'* function of the VETH device before the packets are transmitted on the physical device.

The VETH device also has the Kernel *device* structure and the contents of this structure are initialized when the device is registered with the kernel. The VETH device function pointers such as *hard_start_xmit, init etc* point to the functions implemented in the VETH layer.

The following functions are associated with the VETH layer, the details of which will be explained in the next section:

- **Veth_ioctl:**
  This function receives the ioctl calls from the user-level and based on the type of the ioctl call, it invokes the appropriate function in the VETH layer. The ioctl call can be of three types: create, destroy or list.

- **Veth_create:**
  This function creates the virtual devices, each having a unique interface number and a unique MAC address.

- **Veth_init:**
  This function sets the function pointers of *struct device* for the VETH device.

- **Veth_destroy:**

  This function responds to the ioctl call for deleting a virtual device. It frees the memory occupied by the virtual device to be destroyed.

- **Veth_send:**

  This function receives packets from the IP layer and sends it to the physical layer. Before transmission, the packets are subjected to CBR service and a propagation delay equal to the delay on the satellite link.

- **Veth_receive:**

  This function receives packets from the physical Ethernet layer and sends it up to the IP layer. The packets are de-multiplexed to the right VETH device in the net bottom half queue of the Linux kernel.

- **Veth_change_mtu**

  This function changes the MTU of the VETH device. This function is invoked when *ifconfig* is called to change the MTU of the device.

3.4.2 VETH Detailed Design

The Virtual Ethernet layer is a software layer between the IP and the physical device layer. The virtual devices can be created and configured by different functions of this software layer. This section provides a detailed description of the different functions of the VETH layer, responsible for the functioning of the virtual devices.

3.4.2.1 Ioctl calls from the Control Program

The function *veth_ioctl* is executed in response to the *ioctl ( )* call made by the Control program. The control program makes three requests to the ioctl function:

- *VETH_CREATE_VDEV* -- To create the virtual devices.
- *VETH_DESTROY_VDEV*— To destroy the virtual devices
- *VETH_LIST_VDEV*-- To list the properties

Based on these requests, the *veth_ioctl( )* function invokes the different functions of the VETH layer. The parameters passed by the control program are in the form of a structure. The fields of the structure represent the arguments of the control program for all the options. Program 3.3 represents the contents of the structure.

Program 3.3: Structure *veth_req* for passing arguments to the *ioctl ( )* call

```
Struct  veth_req {
        Char PhyDev[5];  /* name of the physical device,
                         /* Ex : eth0
        Int    itf;      /*  The interface number for VETH
                         /*  device
      Unsigned char srcMac[20];  /* Source MAC address
}
```

3.4.2.2 Creating a VETH device

The function *veth_create ( )* is called from *veth_ioctl( )* in response to the *VETH_CREATE_VDEV* ioctl request. This function takes the following arguments:

- Physical device on which the virtual device has to be created.
- The source MAC address of the virtual device.

The first 3 bytes of the MAC address represents the ITTC KU vendor Id (00:04:86). The interface number assigned to the device is sequentially incremented, starting from 0. After the device interface is created, it is added to *dev_base,* which is an internal linked list in the Linux kernel to keep track of all the devices. This interface is also added to *veth_base*, which is a linked list at the VETH layer to maintain information of all the virtual devices created.

The virtual device keeps track of its underlying physical device (*eth interface)* by storing the name of the physical device in one of the fields of the *veth_device*

structure. The *device structure* related to the physical device can be obtained by this pseudo-code:

```
Physical_device = dev->get(veth_device->phyDevName )
```

The *phy_device* structure mentioned in Program 3.2 keeps track of the list of physical devices and the number of virtual devices that are created on each physical device.

The device flag *vdevCreate* in *struct veth_device* should be set to indicate that the virtual device is created. The device created should be registered as a network device by calling *register_netdev ( ).* This function adds the devices to *dev_base* list and also calls the *veth_init* function to initialize the function pointers of the *struct device* for the VETH device.

Return Value:

Incase, the virtual device is successfully created and initialized, then *veth_create* returns the interface number of the device as a return value to the control program. Otherwise a negative value is returned, in case of error. The control program on receiving the interface value appends it to "veth" and prints the name of the device on the screen. An example of the output incase of successful creation of interface 0 is:

```
Veth0 device created successfully
```

3.4.2.3 Initializing a VETH device

The fields of the *veth_device* structure are initialized in the *veth_create ( )* function, but the function pointers for the *device structure* related to the virtual device are initialized in *veth_init ( ).* The function pointers for the *struct device* point to the respective functions of the VETH layer. An example pseudo-code for initializing the function pointers for *struct device vethDev* of the Virtual Device is listed in Program 3.4.

Program 3.4: Pseudo-code for veth_init function

```
vethDev->open = NULL;
vethDev->stop = NULL;
vethDev->hard_start_xmit = &veth_send; /*Transmit Function
vethDev->mtu = 1500;
vethDev->hard_header_len = 0;
vethDev->addr_len = MAC_LENGTH;
vethDev->change_mtu = &veth_change_mtu;
vethDev->set_mac_address = &veth_set_mac_address;
vethDev->type = ARPHRD_ETHER;          /* Ethernet Type
vethDev->get_stats = &vethDev_get_stats;
vethDev->flags = 0;
vethDev->qdisc = NULL;
vethDev->qdisc_list = NULL;
```

This function also checks whether the veth_device->vdevCreate flag is set or not.

The Linux kernel stores the list of devices having the IP packet types in a variable *ptype_base.* The packet type definitions contain a *func* pointer, which points to the *receive( )* function in the IP layer. That pointer has to be changed to point to the Virtual Ethernet device *receive ( )* function. In case the *veth_device->vdevCreate flag* is set then it searches for all the packet types whose *device structure* points to the physical device on which the virtual device is created. For all those devices, the *packet_type->func* pointer has to be changed from *ip_rcv( ) to veth_recv ( ),* which is the *receive function* for the virtual devices. Before changing the *func* pointer, the original pointer to the IP *receive* function is stored in *phy_device->pdev_recv* function pointer for each physical device. Program 3.5 states the pseudo-code for implementing the change in function pointers.

Program 3.5: Pseudo-code for changing *"packet_type->func"* pointer

```
(for packet_type = ptype_base[ETH_P_IP];
     packet_type != NULL; packet_type->next )

{
  if ( packet_type->dev == phy_device->dev) {

     /* Replace the packet_type->func with veth_recv
function but before that put the original function pointer
in our phy_device->pdev_recv function pointer.  We might
need this back. */

     phy_device->pdev_recv = ptype->func;
     packet_type->func = veth_recv ;

}
```

This initializes the *packet_type-> func* pointer. It points to the *veth_recv function* for the VETH device and the packets, meant for the SBI network, coming on the device driver shall pass through veth*_recv* function before calling *ip_rcv.*

Return Value:

If all the fields get initialized, then the function returns a positive value.

3.4.2.4 Sending Data on the VETH device:

The data traffic has to be routed from the IP layer to the VETH layer before passing it onto the hardware layer. The VETH device does bandwidth limiting and introduces a propagation delay before transmitting the packets. The bandwidth limitation on the connection emulates the satellite transmission link with reserved bandwidth. In the Linux kernel, bandwidth on a link can be limited by implementing Quality of Service (QoS) algorithms for traffic control. QoS in Linux implements different queuing disciplines to provide traffic control. The *device structure* has the provision for

implementing the required specific queuing discipline. The packets coming to the VETH layer are queued according to the queuing discipline specified.

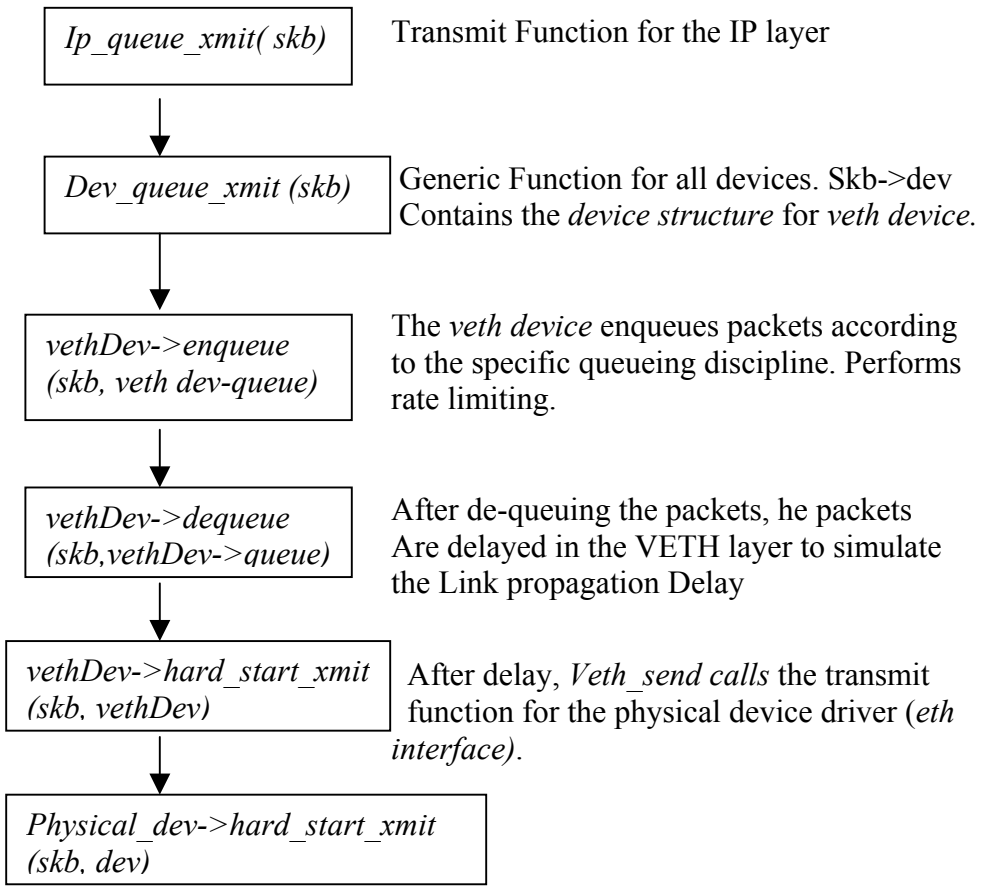
The queuing discipline is specified in the *struct Qdisc* of struct device. The queuing discipline can be specified by the *Traffic Control (tc).* The details of achieving bandwidth limitation are explained in the next chapter.

The packet flow from the application layer to the physical device layer through VETH device is as follows:
The Linux kernel socket implementation routes the data traffic from the application layer through a *write ( )*system call onto the Transport layer and then onto the Network Layer. The network layer for SBI network is the IP layer. The packets from the IP layer call the transmit function for the VETH device and not the physical device.

The packet flow from the IP to the physical device layer is as follows:
The *ip_queue_xmit ( )* is the transmit function in the IP layer which calls *dev_queue_xmit ( )*, which is a generic function for all the devices. The arguments passed to *dev_queue_xmit( )* are the packet contents and the *device structure* for the VETH device. The *struct Qdisc* field of the *device structure* implements the queuing discipline for providing bandwidth limitation. The *dev_queue_xmit* function calls the *enqueue* function to queue the packets according to the specific queuing discipline. The scheduler implements a *qdisc_wakeup () routine* to dequeue the packets and call *veth_send* to transmit the packets to the VETH layer. After the packets come to the VETH layer, the packets are delayed by a *specified* value. The specified value represents the value of propagation delay on the communication link. The network statistics for the number of packets transmitted is incremented in the *veth_send ( ).* Finally, the packets call *dev->hard_start_xmit( ),* where *dev* represents the *device structure* for the physical device (*eth)* and the function invokes transmit function of

the physical device. Program 3.6 states the packet flow from the IP to the physical layer through the VETH layer.

Program 3.6: Packet Flow through the VETH device



3.4.2.5 Receiving Data on the VETH device

As the packets are received on the physical Ethernet device, they are queued to be serviced by the network bottom half at a later time. When the bottom half runs, it checks for the packet type of each received packet and invokes the *receive ( )* function pointed to by *func* pointer for the packet type. When the virtual Ethernet

devices are initialized, the *func pointer* is changed to point to the *receive ( )* function for the virtual device i.e. *veth_recv ( )*.

Whenever a packet is received by *veth_rcv ( ),* its first duty is to find the virtual Ethernet device associated with the destination MAC address, which is located in the header of the received packet. The destination MAC address in the packet header is compared with the *veth_device->srcMac* fields of all the virtual devices that are created on the receiving physical device. Once, the device is found, the function increments the network statistics for the number of packets received and then calls *ip_rcv ( )* for propagation of the buffer up the IP layer. The arguments passed to the *ip_rcv* function are: packet buffer *skb,* device structure for the VETH device *veth_device->vdev,* packet_type.

If the Destination VETH device isn't found, then the packets are not meant for the SBI network and they call the function associated with the *phy_device->pdev_recv( )* function. In this case, the only difference to the arguments of the *receive ()* function is that instead of the *veth device* structure, the *device structure* for the physical device is passed to the function.

The *veth_rcv ( )* function does no processing on the packet and simply forwards the packets to the higher layers.

3.4.2.6 Deleting a virtual Ethernet device

Veth_destroy( ) deletes the specified VETH device. This function is called from *veth_ioctl ( )* in response to the *VETH_DESTROY_VDEV* request.

The argument to this function is the interface number of the VETH device to be deleted. This function searches through the *veth_base* list of virtual devices created. Once the device specified by the argument is found, that device has to be removed

from the *veth_base* and the *dev_base* lists. By calling *unregister_netdevice( ),* the device is removed from the Kernel device list *dev_base.* Since virtual device is a software device occupying the kernel memory, *veth_destroy ( )* frees the memory associated with that device.

This function also decrements the *phy_device->num_vethDev* field when the device is deleted. If the device to be destroyed is the last virtual device for the underlying physical device, then the function pointer *func* for the packet types has to be restored to the original *receive( )* function, which is *ip_rcv ( ).*

```
(For ptype = ptype_base[ETH_P_IP]; ptype !=NULL; ptype->next)
{
      if (ptype->dev == phy_device->phyDev)
            /* Restoring the original pointer of ptype->func */
            ptype->func = phy_device->pdev_recv;
}
```

Return Value:

In case the device is destroyed successfully, the function returns a zero value. The following message shall be printed on the screen if *veth0* has to be deleted:

```
Veth0 destroyed successfully
```

3.4.2.7 Changing the MTU on the virtual Ethernet device

The function *veth_change_mtu ( )* of the virtual device is invoked when the *ifconfig* command is called to change the MTU of the VETH device. During initialization, the function pointer *dev->change_mtu* is set to this function, where *dev* represents the *device structure* for the virtual device. The MTU of the virtual device can be less or equal to the MTU of the underlying physical device.

Chapter 4

4 Constant Bit Rate Service

The previous chapter described requirements and design for constructing the medium for Ethernet communication between two SBI nodes i.e. the Virtual Ethernet Layer (VETH). The VETH layer creates and configures the virtual devices and the connection between the two VETH devices represents the satellite communication link. One of the features of the satellite communication link is providing Constant Bit Rate service (CBR). CBR control provides continuous and dedicated link rate for the established connection. This type of service is useful for transmission of voice, video and data traffic, which requires consistent bandwidth for transmission. CBR service can be provided in Linux by implementing rate limiting. Rate limitation can be implemented by some of the Quality of Service mechanisms like Token Bucket Filter (TBF). QoS disciplines are a part of the traffic control mechanism in Linux.

4.1 Traffic Control In Linux

Linux kernel offers a wide variety of traffic control functions. Incoming packets are examined at the network layer (IP). In case the packets are meant for that node, they are forwarded to the higher layers of the protocol stack for further processing or they are directly forwarded to the network on a different interface. The higher layers also generate data on their own and send them to the lower layers for transmission. The packets that are to be sent out on the output interface for transmission are queued at the respective interface. Traffic control plays an important role here in “queuing” the packets. The traffic control mechanisms can decide if the packets are to be queued or dropped. It can also assign the order in which the packets are sent by prioritizing the packet flows and can also limit the rate of the outbound traffic.

4.1.1 Overview

Traffic control mechanisms provided by Linux, control the flow of packets on a particular device. There are four major components to the traffic control code in Linux kernel:

- Queuing Disciplines
- Classes (within the queuing discipline)
- Filters
- Policing

Each network device has a queuing discipline associated with it, which controls the transmission of the enqueued packets on the device. A very simple queuing discipline would be a *First In First Out (FIFO)* queue. FIFO queuing discipline consists of a single queue, which de-queues the packets in the order in which they are queued. Some of the more complicated queuing disciplines have different *classes* [11]to store the different types of packets. The packets arriving to these queues have to be classified to these different classes by *filters* [11]. Assigning priority to the classes makes it easier to multiplex the traffic of different classes onto the network device. The traffic flow also implements *policing* [11]by discarding packets, which exceed a certain rate.

The *device structure* for each network device has a pointer to *struct Qdisc*, which references the queuing discipline and its function pointers for that device. When the *enqueue* function of the queuing discipline is called, it runs the *filters* one after the other to classify the packet to the *class*. In case, the queue has no classes, the packets are enqueued in the single queue as per the Queuing discipline specified. If the Queuing discipline has classes, then the packets after classification are queued inside the corresponding class by calling the *enqueue* function for the Queuing Discipline "owned" by that class. Usually, when the packets are enqueued, the corresponding

flow can be policed, e.g. by discarding packets which exceed a certain rate or shaped, e.g. by limiting the rate for the outgoing link.

4.1.2 Queuing Disciplines

Each queuing discipline provides set of functions to control its operation. The *struct Qdisc* has function pointers to *enqueue, dequeue, requeue, drop, initialize, reset and destroy* a queuing discipline [11].

In Linux, the packet is enqueued on an interface by calling *dev_queue_xmit( )* function. This function calls the *enqueue ( )* function associated with the queuing discipline for that device. The queuing discipline can be attached to the device through *struct Qdisc* pointer in the *struct device.* If the queuing discipline has classes, then the packet might be classified to different classes and queued inside the class queue. Finally, *qdisc_wakeup( )* function is called to send the packet on the device interface. The *qdisc_wakeup ( )* calls *qdisc_restart ( )* which is the main function to poll the queuing disciplines and send the packets on the device driver. This function calls the *dequeue ( )* function for the packet and invokes the *hard_start_transmit( )* function for the device to transmit the packet.

4.1.3 Classes

Classes represent the medium for differentiating packets for multiple traffic flows. Each class has got its unique identifier and is attached to the root queuing discipline. Classes implement their "own" queuing disciplines, which in turn can create classes and so on.

The various function pointers for each class are stored in *struct Qdisc_class_ops*, which is attached to the *struct Qdisc* of the parent queuing discipline.

4.1.4 Filters

Filters are used by the queuing disciplines to assign the incoming packets to their respective classes. Each filter has a unique identifier and is attached to the parent queuing discipline. Filters are controlled via the function pointers in *struct tcf_proto_ops* in the Linux kernel. There are function pointers to *classify, initialize & destroy* the filters. Filters vary in the scope of packets their instances can classify. Some filters have one instance per queuing discipline to classify packets for all classes. These filters are *generic* [12]. The other type of filters is *specific,* [12] which need more than one instance per queuing discipline to classify the packets.

4.2 CBR Control for SBI Emulation System

This section describes the requirements for implementing CBR control on the SBI connections and the use of Linux Traffic Control mechanisms to implement the same.

4.2.1 Requirements for CBR Control in SBI

In a SBI system, each SBI node is capable of handling different types of data traffic, example: Telemetry, Observational Data etc. The SBI data traffic models the actual data collected by the satellites and relayed to other satellites and ground stations. The nature of the data traffic through the SBI emulation nodes might be continuous or periodic. The SBI nodes representing the relay satellites transmit and receive data continuously, while the nodes emulating the observational satellites have some links dedicated for data collection and some links for relaying data. The ground station nodes act as destinations to receive data traffic and also as relay stations to route data to other nodes.

The SBI node should be able to utilize the node bandwidth efficiently for transmitting and receiving different types of data. The node has multiple connections through its virtual devices. Each link should have dedicated bandwidth for transmitting data

traffic and this can be implemented by rate limitation mechanisms on the link, provided in Linux kernel.

The Linux Kernel implements traffic control mechanisms through Quality of Service queuing disciplines. One such discipline is Token Bucket Filter (TBF)[13] mechanism, which provides rate limiting on the connection. In case of multiple traffic flows through the same link, the Class Based Queuing (CBQ)[16] can be applied to the virtual device. CBQ creates classes for different flows.

4.2.2 CBR Control Mechanism

CBR control can be provided at each SBI node by TBF queuing mechanism. Each virtual Ethernet device implements TBF queue. The parameters for creating the queue will be specified by a utility called as "tc", which stands for Traffic Control. The details of the TBF mechanism are explained in the next section.

Each link has a certain data rate. The relay satellites have high data rates, while the data source satellites have low data rates. According to the type of entity, the SBI node represents the link rate shall be limited by TBF mechanism. In case of multiple connections through the same node, the link rates shall be allocated as per the requirements, with the sum of the total link rates not exceeding the available bandwidth on the node. In case of any new link connection, the bandwidth for that link will be assigned after considering the current bandwidth utilization and the bandwidth requirements for the new connection.

In case of multiple connections through the nodes, then device implements Class Based Queuing (CBQ). CBQ creates classes for different traffic flows. Filters such as U32 filters are used to classify the packets to different classes on the basis of different destination and source IP addresses. The details of CBQ and U32 filters are explained in the later sections.

4.3 Token Bucket Filter (TBF)

4.3.1 TBF Mechanism

Token Bucket Flow (TBF) passes packets arriving at rate in bounds of some administratively set limit. The TBF implementation consists of a buffer (bucket), constantly filled with virtual pieces of information called tokens, at a specific rate (token rate). The most importance parameter of the bucket is its size, that is the number of tokens it can store.

Each arriving token lets one incoming data packet out of the queue and is then removed from the bucket. There are three possible scenarios with different values of the token generation rate and the incoming data rate:

- The data packets arrive into TBF queue at a rate *equal* to the rate of the incoming tokens. In this case each incoming packet will have a token and so will not be delayed for transmission.
- The data packets arrive at a rate *smaller* than the token rate. So a fewer tokens will be removed from the bucket with the packets. This would lead to accumulation of tokens up to the bucket size. The saved tokens could be used to send data over the token rate, if short data burst occurs.
- The data arrives into TBF at a rate *bigger* than the token rate. In this case, the data packets can be sent only until all the tokens accumulated in the bucket are used. After that, over limit packets are dropped.

The last scenario is important because it allows shaping of the bandwidth available to the data packets. Shaping the bandwidth causes the link bandwidth to be limited to a particular rate.

4.3.2 Linux TBF Queuing Discipline

There are three basic parameters for each TBF queuing discipline:

- *Rate* – The rate at which the tokens are generated in the bucket. The rate represents the average transmission rate for a traffic flow.
- *Bucket size or Burst Size* -- The number of tokens that the token bucket can store. Every token is equivalent to one byte.
- *Limit*—Limit represents the sum of the queue size and the bucket size.

The limit parameter for the TBF decides whether the packets should be policed or shaped. If the limit is equal to the bucket size the queue size is zero and the over limit packets are dropped. This polices the data stream. If the limit is greater than the bucket size, the over limit packets are queued, which shapes the stream. Therefore, the Linux TBF queuing discipline is a meter, shaper and policer, all in one.

4.4 Class Based Queuing

Class Based Queuing (CBQ) [16]discipline is used to limit the outgoing bandwidth on the link. It provides a very flexible and efficient approach to first classifying the user traffic and then assigning bandwidth characteristics to each traffic class. Each class represents an individual traffic flow on the device. Each traffic class can be assigned a committed bandwidth rate, which is a part of the total bandwidth allocated on the device. The link rate for each class can be fixed i.e. bounded or flexible, which means that the link can borrow bandwidth from its parent class or queuing discipline.

Whenever the packets arrive at the root queue, they are classified. If the traffic class hasn't used all of its bandwidth i.e. the class is under limit, the packet flows immediately to the outbound link and no rate limiting is required. But if the class is over limit, then the packet is placed in its queue and is rate shaped onto the outbound

link. The packet might be allowed to “borrow” from the parent bandwidth if the class to which the packet is classified is not “bounded”.

4.5 U32 Classifier

The U32 classifier [12]is the most advanced filter available in the current implementation of filters. The U32 classifier contains 2 fields: a *selector* and an *action*. The selectors try to match the packets according to the different fields of the packet header. It performs certain actions once the packet matches to the selector criteria. One of the actions is to direct the packet to its associated class after classification.

The U32 *selectors* define the pattern to match the packet contents with the mask and the offset for the start of the pattern matching. The pattern mostly consists of the fields in the packet header, which classify the packets. The fields might be of the Protocol layer header such as IP header or the Transport Layer header, which is the TCP/UDP header. Some of the patterns used for matching are:

- Destination IP address
- Source IP address
- Source, Destination Port.

4.6 “tc” Utilty

“tc” [14]stands for Traffic Controller, which is a user level program to create and associate queues with the output devices. It can be used to set up various queues and associate classes with each of those queues. It also configures the filters for classifying the packets into the classes. The usage for “tc” [15] is:

```
tc [OPTIONS] OBJECT {COMMAND | help}
where, OBJECT := { qdisc | class | filter }
       OPTIONS := { -s [statistics] | -d [details] |
                    -r [raw] }
```

The Object could be queuing discipline, class or a filter.

4.6.1 Interface between the user and the Linux kernel

The interface for the "tc" utility between the user and the Linux kernel traffic elements is achieved using netlink sockets [12]. The interface information is specified in the files *pkt_cls.h* and *pkt_sched.h* inside the Linux kernel. *Rtnetlink*, which is based on netlink is used to exchange the traffic control objects between the user level and the kernel level. The netlink sockets use *struct sockaddr_nl* address structure, which is used by the user level code to communicate with the kernel. Whenever the "tc" command is executed for a specific action, a *sendto* is done on the netlink socket. The *rtnetlink_rcv_msg* function in rtnetlink.c receives the message from the user space. It examines the message header to determine the message type. Depending on the message type, the corresponding function is invoked.

4.7 CBR control in SBI Emulation System

The *Communication Emulation* unit, which is a part of the SBI emulation software on the SBI nodes, creates the Virtual Ethernet devices to connect to the other SBI nodes. The control program module for configuring the VETH layer is a part of the *Communication Control Unit.* The *Communication Control* unit also hosts the "*Traffic Control (tc)"* utility, which configures the traffic control modules in the Linux Kernel.

A communication link represents an IP over Ethernet connection between two virtual Ethernet devices. The link rate represents the data rate of the satellite that is transmitting traffic. The data rate capacity of the link can be limited to the required data rate by using Token Bucket Filter (TBF) queuing mechanism. The *tc utility* is loaded on the SBI nodes and is a part of the *Communication Control Module* and configures the parameters of the Queuing disciplines for each of the virtual devices.

The parameters for each of the virtual devices are passed from the Operations Node via the Emulation Manager.

Quality of Service can also be provided at each link by the traffic control disciplines. QoS service deals with classification of different data traffic into different *classes* inside the *queue* implemented for the communication link. The Linux scheduler schedules the transmission of packets from these different classes depending on the importance of the data classified.

Figure 9 shows the interaction between the Emulation Modules for setting the CBR control on each link. The emulation manager receives the QoS parameters regarding the Queuing discipline and the data rate values from the Operations Node.

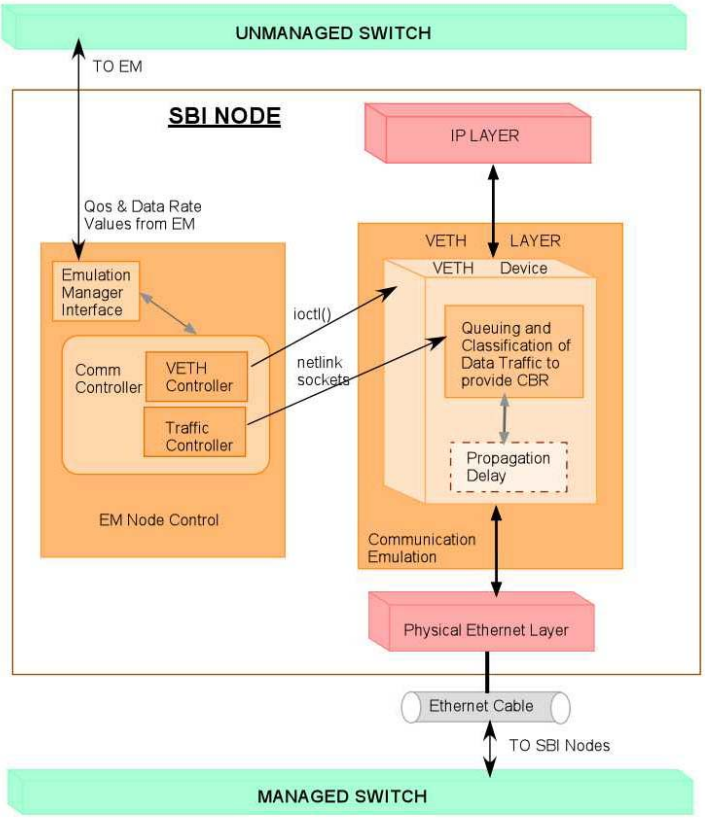


Fig 9: SBI Node Controls for providing CBR service

The *veth_controller* module creates and configures the virtual devices and the *traffic controller* module executes the "tc" command through the netlink sockets to set up the CBR control on the virtual device.

The parameters received from the Emulation Manager are passed to the Traffic Controller module. The traffic controller module attaches the specified queuing discipline to the virtual device. If the virtual device has just one type of data traffic flowing through it, then the Queuing discipline need not have classes and can implement a single Token Bucket Filter through it. In case the virtual device is handling different types of data traffic, Example the relay satellites, then Class Based Queuing has to be implemented and packets have to be classified by filters. Each of the class queues would then implement a Token Bucket Filter Queuing Discipline, which would provide CBR control as per the specifications. The details of providing CBR control for a VETH devices with or without Class Based Queuing are explained in the next section.

4.7.1 Example Scenario for CBR Control

To demonstrate the CBR control for the SBI system, a scenario is designed which takes into account all the different types of nodes in the SBI system. The CBR control for each link depends upon the amount of data traffic handled by that node. The need for differentiating traffic into classes arises if a node is handling different types of traffic with different degrees of importance. Figure 10 shows a probable SBI scenario, which provides CBR Control on the links.

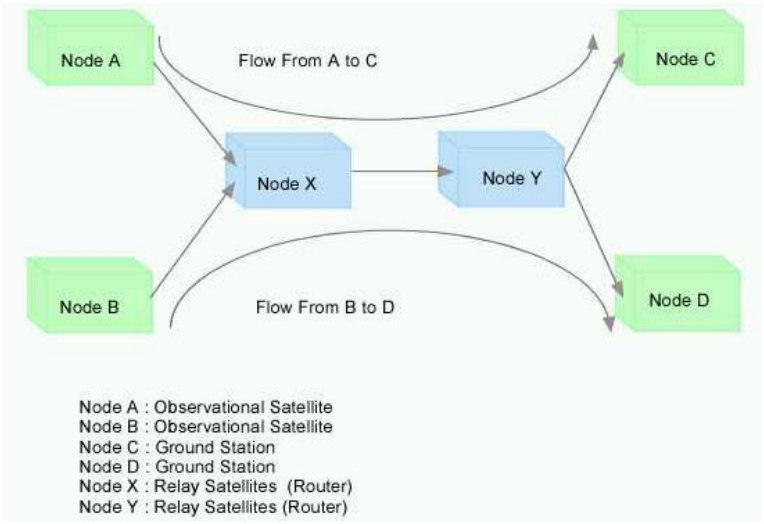


Fig 10: SBI System Scenario implementing CBR Control

Nodes A and B represent observational satellites and the links, which are dedicated to data collection and transmission to the ground stations or other satellites. These links do not relay data from one node to other. These links just have one type of traffic flow through them and so they implement CBR control without Class Based Queuing.

Nodes C and D represent ground stations, which receive the observational data from the satellites A and B via the router satellites. The ground station nodes are the destination nodes in this scenario and don't implement CBR control

Nodes X and Node Y represent the High data rate Relay satellites, which relay data for different destinations. Node X has traffic for different destinations through its link from X to Y. Node X implements CBR control with Class Based Queuing with classes created for different destinations. Node Y has two separate links for the 2 destinations and so doesn't create classes for the link. In case, there were multiple data flows through Node Y on the same link, then classes would have to be created.

4.7.2 CBR control without Class Bases Queuing (CBQ)

Consider data traffic flow from node A to node X. The virtual devices on both the nodes form a connection link. The link rate on this connection has to be equal to the data rate of the transmitting node. The link rate can be limited to the required value by implementing TBF queuing discipline.

Each node receives its QoS parameters from the Operations Node via the Emulation Manager. The Operations node decides on the routing tables for each of the nodes. The routing tables denote the destination and the gateway IP addresses and the virtual device on which to send the traffic. In case of a single traffic flow through a virtual device, there would be only one route associated with that device. The Operations node decides on the link rate for that device connection and then sends the parameters accordingly to the Emulation Manager. The Emulation Manager transmits the parameters to the node, which sets the TBF queue according to the parameters.

For Example, if the connection from virtual device *veth1* of Node X to *veth2* of Node A has to be limited to around 450kbps, then the *tc command* would look like:

```
tc qdisc add dev veth1 handle 10: root tbf 450kbit
Burst 450kb/8 limit 450kb

Where, qdisc = queuing discipline
       Handle = identity for the queuing discipline on
                That node
       Root = specifies that it is a root queue.
       Burst = specifies the size of the largest burst
       Limit = burst + queue size.
```

The parameters sent by the Operations node to the Traffic controller module:

- Virtual Device on which TBF queue has to be set

- Rate at which the link rates should be limited. This represents the data rate for the transmitting node. The transmitting node might be a satellite or a ground station and the rates mean the data rates for these entities. The data rates for the satellites are obtained from the configuration files on the Emulation Manager.
- Burst Size for the TBF queue, which will be proportional to the link rate.
- Limit represents the sum of the burst size plus the queue size.

The *Traffic Controller* module on receiving the parameters calls the "*tc"* utility to create the TBF queuing discipline for the virtual device.

4.7.3 CBR Control with Class Based Queuing (CBQ)

CBQ can be created on the device having multiple flows. The criteria for classification could be traffic flows meant for different destinations and source IP addresses.

Creating a CBQ discipline for a device requires the configuration of three components:

- Queues
- Classes
- Filters

The CBQ queuing discipline is the root queue for the virtual device, which creates classes for different traffic flows. Filters have to be created to classify the packets to these different classes.

Referring to Figure 10 for the scenario, node X has two different flows from node A and node B. The flows are meant for different destinations nodes C and D respectively. So Node X creates a Class based Queuing Discipline for the virtual

device. CBQ has 2 classes for the 2 traffic flows. Packets arriving at the root CBQ queue are classified according to the destination IP address and the source IP, which is the physical device on which the packet came. U32 filters can be used to classify the packets to their classes. The classes can have simple FIFO queues because the rate limitation for the traffic is done at the node on which the traffic is generated. The virtual device might be allowed to use the whole bandwidth or it might be allocated some bandwidth from the total node bandwidth. In this case, the root class for the CBQ created has to have the rate restricted to the allocated bandwidth.

Figure 11 illustrates CBR control on the SBI nodes using Class Based Queuing (CBQ).

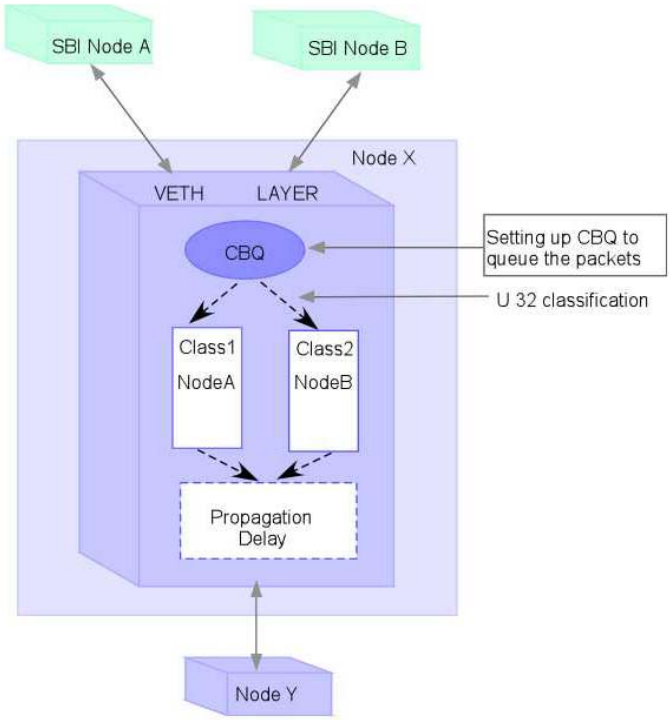


Fig 11: SBI Node CBR Control with Class Based Queuing

The Operations Node decides the need for CBQ queuing for the SBI node. Accordingly, the parameters for the CBQ queue and the parameters for the individual classes and their queues are passed from the Operations Node to the concerned node through the Emulation Manager.

The parameters passed by the Operations Node to the SBI node for CBQ:

- The total bandwidth on the virtual device,
- The device name on which to create CBQ.

The parameters for creating classes:

- Root Class attached to the root queuing discipline.
- Data rates for each class
- Individual queuing disciplines for the class: FIFO queuing discipline.

Filters are created for each class. The filters implemented at the root of the CBQ classify the packets on the basis of the destination address. The filters implemented are U32 filters.

The parameters passed from the Operations Node for creating the filters:

- The virtual device
- The destination IP address and the source IP addresses.

On receiving the parameters for all the components, the Traffic Controller module runs the "*tc" utility* script for the specified virtual device. This utility configures the CBR control for the device. A complete example is explained in the APPENDIX B. But a short example is provided below:

```
Creating a root CBQ queue and CBQ class on device veth1 on
Node X with device bandwidth as 10 Mbps and the root class
bandwidth restricted to 10Mbps.
```

```
tc qdisc add dev veth1 root handle 10: cbq bandwidth 10mbit
avpkt 1000 allot 1514 cell 8 mpu 64

tc class add dev veth1 parent 10:0 classid 10:1 cbq bandwidth
10mbit rate 10mbit allot 1514 weight 100kbit prio 2
```

The first command creates a root queue

Where, avpkt: 1000 bytes

Allot: the size of the Ethernet MTU plus the Ethernet header (14bytes).

Mpu: minumum number of bytes sent in one packet

Cell: the boundaries of the bytes in the packets Transmitted.

Rate: the bandwidth allocated to the class

Bandwidth: Maximum bandwidth available to the device

The second command creates a root class attached to the parent queue. The variable “prio” stands for priority assigned to the class. Further, within the root class, there would be classes created for the 2 flows. The entire example is explained in the APPENDIX B.

Chapter 5

5 Link Propagation Delay

The previous chapter provided explanation for having CBR control on the Emulation Communication Satellite link. The CBR control limits the rate on the link to the data rate of the satellite. So the satellites can route their data as they collect, through the virtual device connections to other satellites and ground stations. Since the distance between the satellites or satellites and ground stations is very large, the data transmission on the communication link suffers from very high propagation delays. On account of high propagation delays, large amount of data would be *in-flight* on the transmission link [4].

The Communication Emulation Unit of the SBI System models the features of the communications links. The last chapter talked about emulating CBR control through Quality of Service mechanisms. This chapter details out the method for simulating the propagation delay on the emulation link. The propagation delay is introduced in the Virtual Ethernet (VETH) layer after the CBR control. The packets are delayed before being transmitted on the physical device.

The chapter first starts with the requirements for simulating the propagation delay and proceeds to state the analysis done for calculating propagation delays on some of the actual satellite transmission links. Based on some of the analysis results, this chapter further describes the algorithm for simulating the propagation delay at the VETH layer.

5.1 Requirements for Simulating the Propagation Delay

The Communication Emulation Unit on the SBI nodes emulates the communication system on the satellites and ground stations in an actual satellite system. The satellites are at different altitudes on the earth and the distance between them and the ground

stations is very large. The propagation delays on the transmission link can be as high as 250ms for satellites in GEO orbits. On account of such high propagation delay, large amount of packets are in flight on the link before they reach the destination.

- The SBI Emulation System emulates these transmission links as Ethernet connections between two virtual Ethernet Interfaces. The propagation delay on these Ethernet connections is very negligible on account of very small distances. The packets suffer from transmission delay, which depends on link bandwidth and packet size and queuing delay, which is varying depending on the amount of data queued.

- To model the communication link, simulation of propagation delay is very important. The propagation delay is in milliseconds, which is far more than the transmission delay. Since the transmission delay is lesser than the propagation delay, the packets have to be queued at the VETH device before they are transmitted on the device. The amount of time the packets need to be queued would correspond to the propagation delay on the transmission link.

- On account of high propagation delays, there are many packets in-flight along the transmission path until the first packet reaches the destination. The numbers of in-flight packets have to be queued at the VETH layer before the first packet is transmitted from the VETH layer.

- Therefore, one of the requirements is to find out the queue size that would be necessary to place the packets until they are delayed. In case the queue size is not up to the requirements, the packets will be dropped at the VETH layer in case if the queue is completely filled. Since the propagation delay varies in different satellite links, each connection can have different queue size. The queue size has to be calculated and allocated at the VETH layer.

- Also, on a particular link, the propagation delay varies, which varies the queue size on the VETH layer. The variations in the queue size have to be known so as to change the queue size according to the changes in the propagation delay.

- The propagation delay at the VETH layer represents the value on the actual link. During the simulation time, this value might change as the distance between the two elements forming the link change. The changes in the delay values should be notified to the VETH layer. This requires a control program that shall interface with the delay functions in the VETH layer. The details of the implementation are explained in the later sections.

Based on these requirements, an analysis has to be first made for calculating the propagation delay values on actual satellite transmission links. Satellites at different orbits should be considered. Calculation of propagation delays on the actual transmission links can then be considered for the analysis. The values obtained for different type of links will help provide a better picture about the maximum and minimum delay values and the delay variations for a particular link.

5.2 Propagation Delay Variations in Satellite Systems

As according to the requirements, one of the tasks is to find the range of values for the propagation delays normally occurring on the transmission links. Since propagation delays depend on the distance between the elements, the propagation delays between all satellite links aren't the same.

For a complete analysis of the delay variations on a transmission link, all the different types of satellites and ground stations have to be considered. There are three types of satellites, satellites in LEO, MEO and GEO orbits. All the possible links between the satellites and also between the ground stations have to be taken into account for delay

computations. Therefore, a total of six different types of links can be considered for a thorough analysis.

5.2.1 Using Satellite Tool Kit for delay analysis

Satellite Tool Kit (STK) is an analysis tool that addresses all phases of the satellite systems. STK models the satellite systems and performs analysis of different properties of satellites and facilities. These properties include vehicle propagation, determination of visibility areas and times for satellite connections, computation of access times for the transmission links (Line of Sight) and propagation delay, display of orbital positions and generation of results in textual and graphical formats. Based on simple inputs through shell scripts or command lines, STK generates orbital paths for a variety of space and ground based objects, such as satellites and ground station facilities [17].

Scenarios can be created in STK, where the scenario elements are the actual satellites and ground stations obtained from the STK satellite database. The STK can simulate the entire mission of the satellites through its graphical user interface. It also provides reports and graphs on various properties of the satellites and facilities throughout the simulation time.

STK has a Connect module to provide the user an easy way to connect with STK and work with it in a client-server environment. The interface is using TCP/IP or domain sockets. Third-party applications can connect to STK using the library provided with the STK Connect Module. This library contains functions, constants and other messaging capabilities that help the user use STK. Connect module also allows the user to modify the standard messaging formats also.

Figure 12 shows the interaction of the Connect module[17] with the user applications to use STK libraries and control simulation graphical interface.

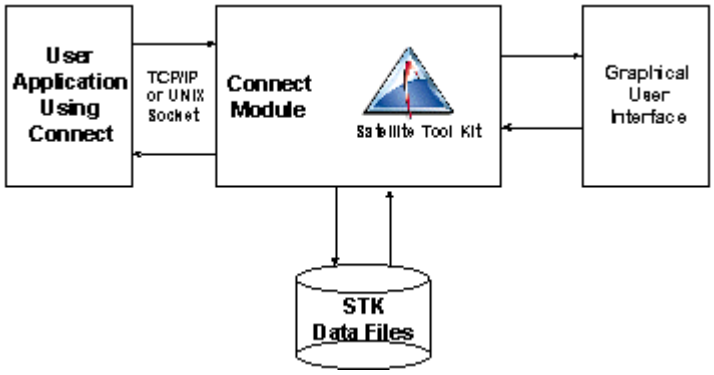


Fig 12: User Application using Connect Module to interface with STK

The Connect Module has a list of commands, which the user can execute to control the simulation and obtain results. These commands can be executed on the command line or through a file.

To analyze the propagation delays in satellite systems, a scenario having 3 satellites (all in different orbits) and a ground station was simulated. This scenario covered all types of possible satellites as each satellite is in a different type of orbit. Propagation delays were calculated for all the links possible between the elements and a Report stating the values of propagation delay versus simulation time was obtained. The exact details are covered in the next section

5.2.2 Scenario Details

The scenario to be simulated had four elements, three satellites and a ground station. All the satellites were in different orbits, LEO, MEO and GEO. The satellites were obtained from the STK database and the ground station Facility was specified by the latitude and longitude on the earth.

Elements in the scenario:

1) **Facility:**

Name: Ground Station

Latitude: 12 Deg.

Longitude: 45 Deg.

2) Satellites:

	GFO_LEO	**USA_144_MEO**	**SKYNET_4E_GEO**
Type	Satellite	Satellite	Satellite
Official Name	GFO	USA 144	SKYNET_4E
Mission	Oceanography	Radar Imaging	Military
Apogee	789Km	3131Km	35797Km
Perigee	783Km	2689Km	35777Km
Period	100.5min	148.5 min	1436.0min
Inclination	108.1deg	63.4deg	2.5deg
Orbit	LEO	MEO	GEO

Table 1: Details of the Satellite elements in the Scenario

The STK tool loads these elements in the simulation. The graphical interface shows these elements and their orbits. The GEO satellite is stationary above the ground station. The LEO and the MEO satellites are moving in opposite directions to each other. The simulation time period was fixed to be 1 day, which is 1440 minutes.

The STK software can compute a report of the access times between two elements. Since all the possible links were to be considered, there were six different links established:

1) Ground Station –LEO 2) Ground Station-MEO

3) Ground Station-GEO 4) MEO –GEO

5) MEO-LEO 6) LEO-GEO

An STK Report was generated giving the access times for each of the links. Also, propagation delay on each link was calculated for every minute of the simulation as the simulation time was in minutes.

5.2.3 Analysis of Link Propagation Delays

For each access time frame, a graph of the propagation delay values against the simulation time was generated. The graphs were generated for all the links. Each of the links is discussed below.

5.2.3.1 Link between Ground Station and a LEO satellite

Data

Satellite: GFO_ LEO

Ground Station: Facility

The STK generates a report for the Access times for the link during the entire simulation.

STK Access Report:

Access	Start Time (Epoch Minutes)	Stop Time (Epoch Minutes)	Duration (Minutes)
1	70.7942	80.2817	9.487
2	166.4119	181.2301	14.818
3	267.8557	275.1652	7.31
4	737.0386	739.7967	2.758
5	828.6642	843.2451	14.581
6	928.5839	939.8317	11.248

Table 2: STK Access Report for a LEO-Ground Station Link

Graphs:

The Delay variations are observed to be similar for all the accesses and so only a few of the graphs are shown.

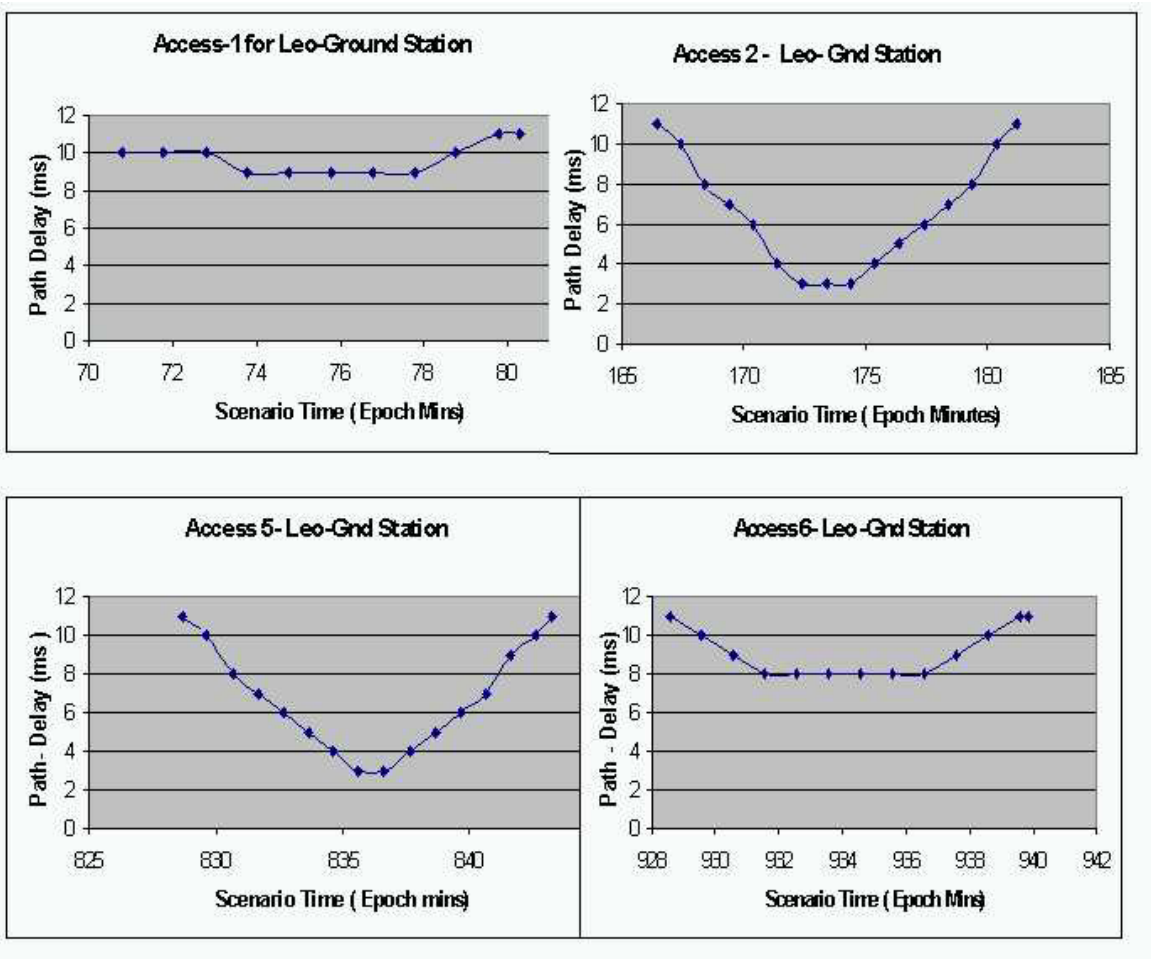


Fig 13: Propagation Delay versus Time for LEO-Ground Station Access

Observations:

- The maximum delay on a LEO-ground station link = 11ms
- Minimum Delay = 3ms
- The ground station can access the satellite for around 15 minutes maximum during each access.

- Maximum delay variations per minute are 2ms.

5.2.3.2 Link Between Ground Station and MEO Station

Data:

Satellite: USA_144_MEO

Ground Station: Facility

There are 6 accesses during the entire simulation. The table below states the Access Report for the Ground Station-MEO link.

STK Access Report:

Access	Start Time (Epoch Minutes)	Stop Time (Epoch Minutes)	Duration (Minutes)
1	121.344	159.444	38.1
2	276.7916	314.3661	37.574
3	626.3372	646.9397	20.602
4	776.8671	817.618	40.751
5	932.7947	971.2767	38.482

Table 3: STK Access Report for a MEO-Ground Station Link

Since all the access times are similar, the graphs generated are shown for only four of the access times.

Graphs:

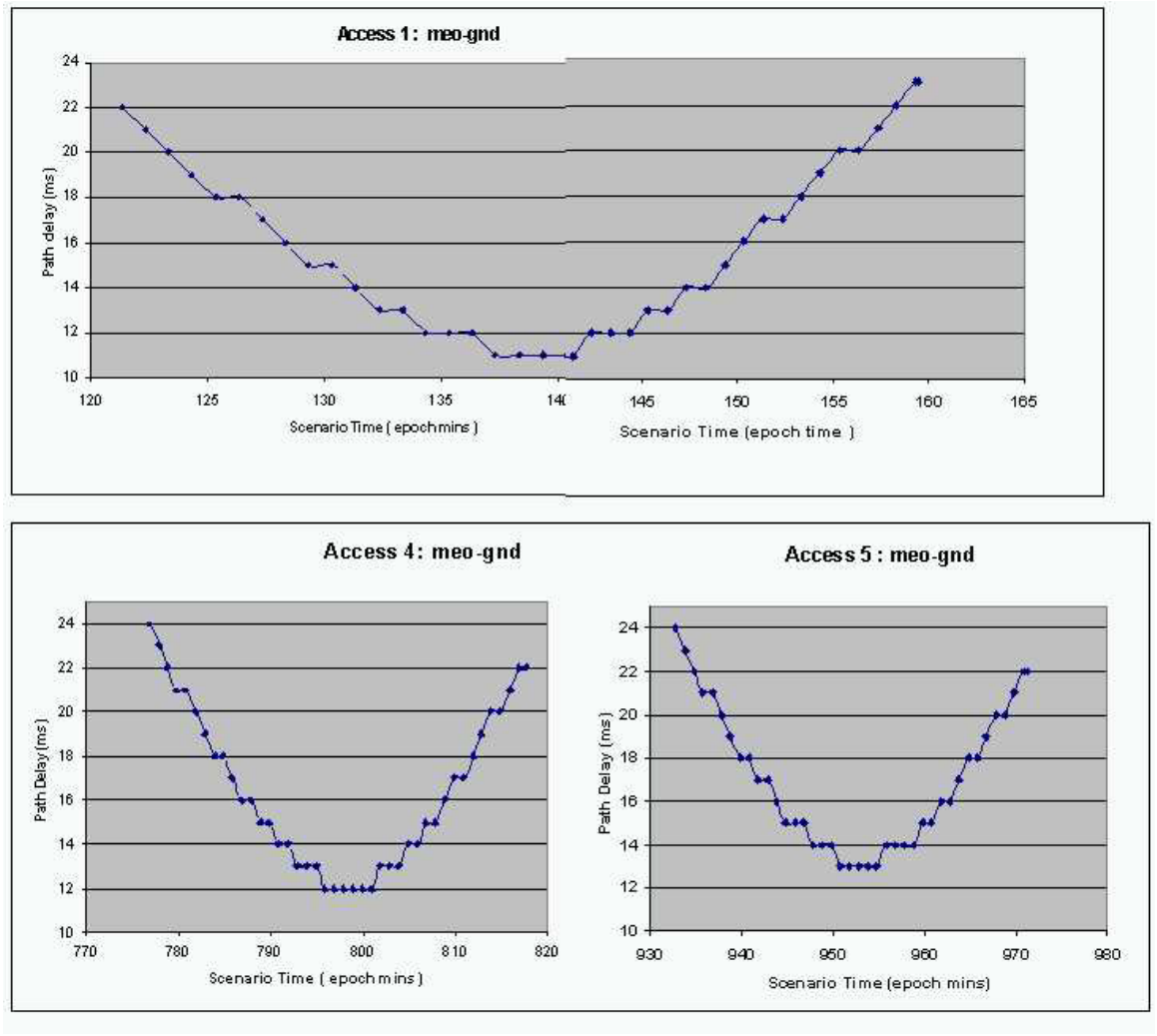


Fig 14: Propagation Delay versus Time for MEO-Ground Station Access

Observations:

- Max delay = 24ms
- Minimum Delay = 11ms
- The ground station can access the MEO satellites for 40 minutes for each access duration.

- Maximum delay variations per minute: 1ms. The delay doesn't change with every minute, its constant for some amount of time before changing.
- The delay variations for each access are the same with slight variations in the maximum and the minimum values.

5.2.3.3 Link Between Ground Station and GEO Satellite

Data:

Satellite: SKYNET_4E_GEO

Ground Station: Facility

The GEO synchronous satellite appears stationary above the ground station and so the ground station can access the satellite continuously throughout the simulation.

STK Access Report:

Access	Start Time (Epoch Minutes)	Stop Time (Epoch Minutes)	Duration (Minutes)
1	0	1440	1440

Table 4: STK Access Report for a GEO-Ground Station Link

The graph of propagation delay versus the scenario time (epoch minutes) is plotted as follows:

Graphs:

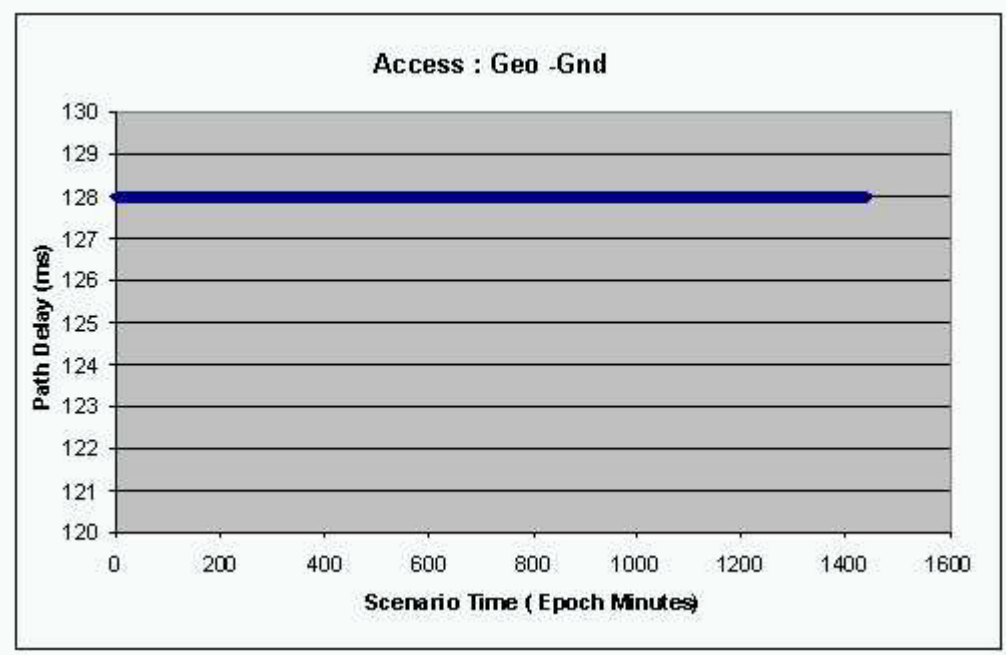


Fig 15: Propagation Delay versus Time for GEO-Ground Station Access

Observations:

- Since the GEO satellite is stationary with respect to the ground station at all times, the delay is constant throughout the simulation period.
- Delay = 128ms

5.2.3.4 Link between the LEO and MEO satellites

Data:

LEO Satellite: GFO_LEO

MEO Satellite: USA_144_MEO

The two satellites are travelling in the opposite direction towards each other. The number of accesses on the link is more and the duration of each access is uniform 25

minutes. The STK Access Report generates the access duration for 24 accesses during the entire simulation period.

STK Access Report

Access	Start Time (Epoch Minutes)	Stop Time (Epoch Minutes)	Duration (Minutes)
1	28.2416	53.7676	25.526
2	89.0692	113.006	23.937
3	148.223	173.1477	24.925
4	208.9476	233.5161	24.569
5	268.6372	292.8469	24.21
6	328.3106	353.802	25.491
7	389.0586	412.9847	23.926
8	448.2038	473.2368	25.033
9	509.0033	533.4735	24.47
10	568.5956	592.867	24.271
11	628.3841	653.8252	25.441
12	689.0444	712.964	23.92
13	748.1949	773.3254	25.131
14	809.0523	833.4306	24.378
15	868.5514	892.8966	24.345
16	928.4606	953.8382	25.378
17	989.0261	1012.945	23.919
18	1048.197	1073.412	25.215
19	1109.095	1133.388	24.293
20	1168.506	1192.935	24.429
21	1228.538	1253.842	25.304
22	1289.003	1312.929	23.925

23	1348.21	1373.494	25.285
24	1409.132	1433.345	24.214

Table 5: STK Access Report for a LEO-MEO Link

The graphs are plotted for each of the access duration. The delay variations for all the access graphs is found to be similar and so only a few graphs are shown here to denote the delay variations.

Graphs

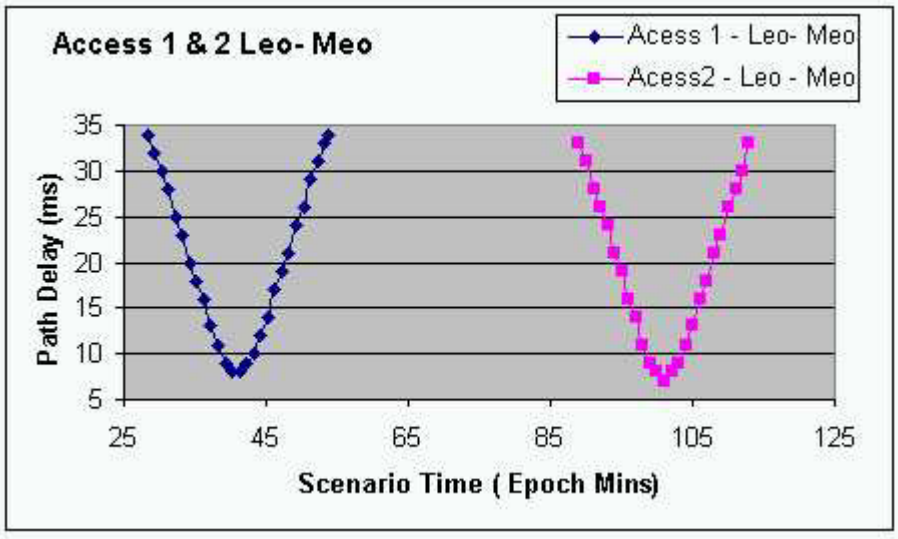


Fig 16: Propagation Delay versus Time for LEO-MEO Access

Observations:

- Since the 2 satellites are travelling opposite to each other, there are more accesses.
- Maximum delay = 34 ms
- Minimum delay = 7 ms
- Maximum Delay Variations per minute = 3ms
- Total access time for each duration = 25 minutes.

- The delay variations exhibit a same pattern for each access. The initial delay is around 34ms and then later on decreases gradually till approximately 7ms. As the satellites go further away from each other, the delay increases to the maximum value of 34ms till the loss of line of sight.

5.2.3.5 Link between the GEO and MEO satellites:

Data:

GEO Satellite: SKYNET_4E_GEO

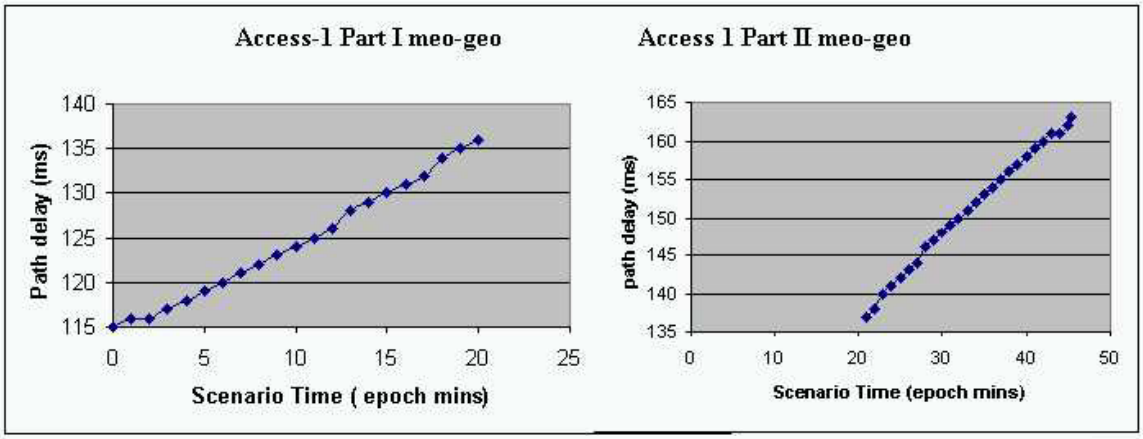
MEO Satellite: USA_144_MEO

STK Access Report:

Access	Start Time (Epoch Minutes)	Stop Time (Epoch Minutes)	Duration (Minutes)
1	0	45.4247	45.425
2	87.9136	556.834	468.92
3	596.4867	709.072	112.585
4	752.7151	871.6173	118.902
5	899.9558	1227.519	327.563
6	1261.964	1376.87	114.906
7	1421.919	1440	18.081

Table 6: STK Access Report for a MEO-GEO Link

Graphs:



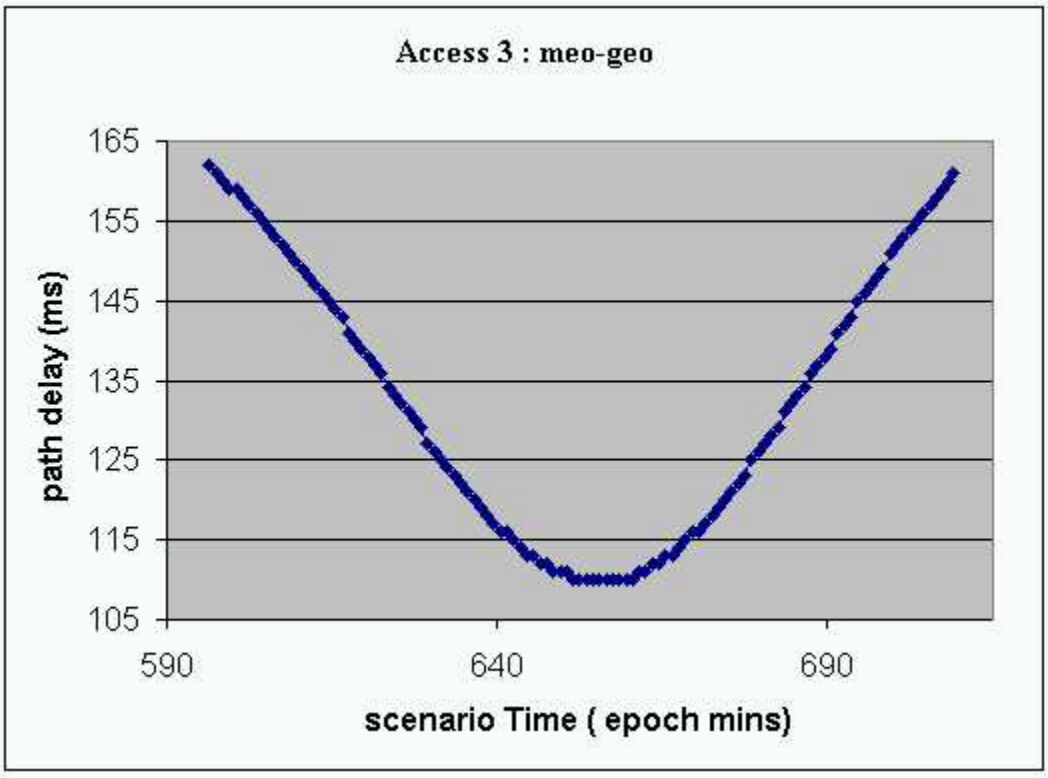


Fig 17: Propagation Delay versus Time for MEO-GEO Access

Observations:

- Maximum Delay = 163ms
- Minimum Delay = 112 ms
- Maximum Delay Variations per minute = 2ms
- The delay variations aren't same for all the accesses. For access intervals more than 300 minutes (Access2), the delay variations are uniform with 1ms variation per minute.
- For access intervals of 110 to 115minutes (Access 3) , the delay variations are uniform (1ms variation per minute), but a slight change in the pattern is observed at delay values between 145 and 135 ms. At around 142 ms, the delay variation is 2ms per minute.
- Similar pattern is also observed for access intervals lesser than 45 minutes (access 1). A slight variation in delay pattern is observed between the delay values 150ms and 140ms.

5.2.3.6 Link between the LEO and GEO Stations

Data:

GEO Satellite: SKYNET_4E_GEO

LEO Satellite: GFO_LEO

STK Access Report:

Access	Start Time (Epoch Minutes)	Stop Time (Epoch Minutes)	Duration (Minutes)
1	0	6.2775	6.278
2	44.7646	105.911	61.146
3	141.0331	206.5725	65.539
4	230.7235	381.4226	150.699

5	400.6803	468.1245	67.444
6	501.9027	563.581	61.678
7	601.682	661.6527	59.971
8	700.7358	760.8007	60.065
9	798.604	860.6509	62.047
10	893.4923	962.2723	68.78
11	978.1531	1129.955	151.802
12	1156.339	1220.974	64.635
13	1256.744	1317.613	60.869
14	1356.279	1416.128	59.849

Table 7: STK Access Report for a LEO-GEO Link

Graphs are plotted for Access 3 and Access 4 since there are only two variations in the access durations.

Graphs:

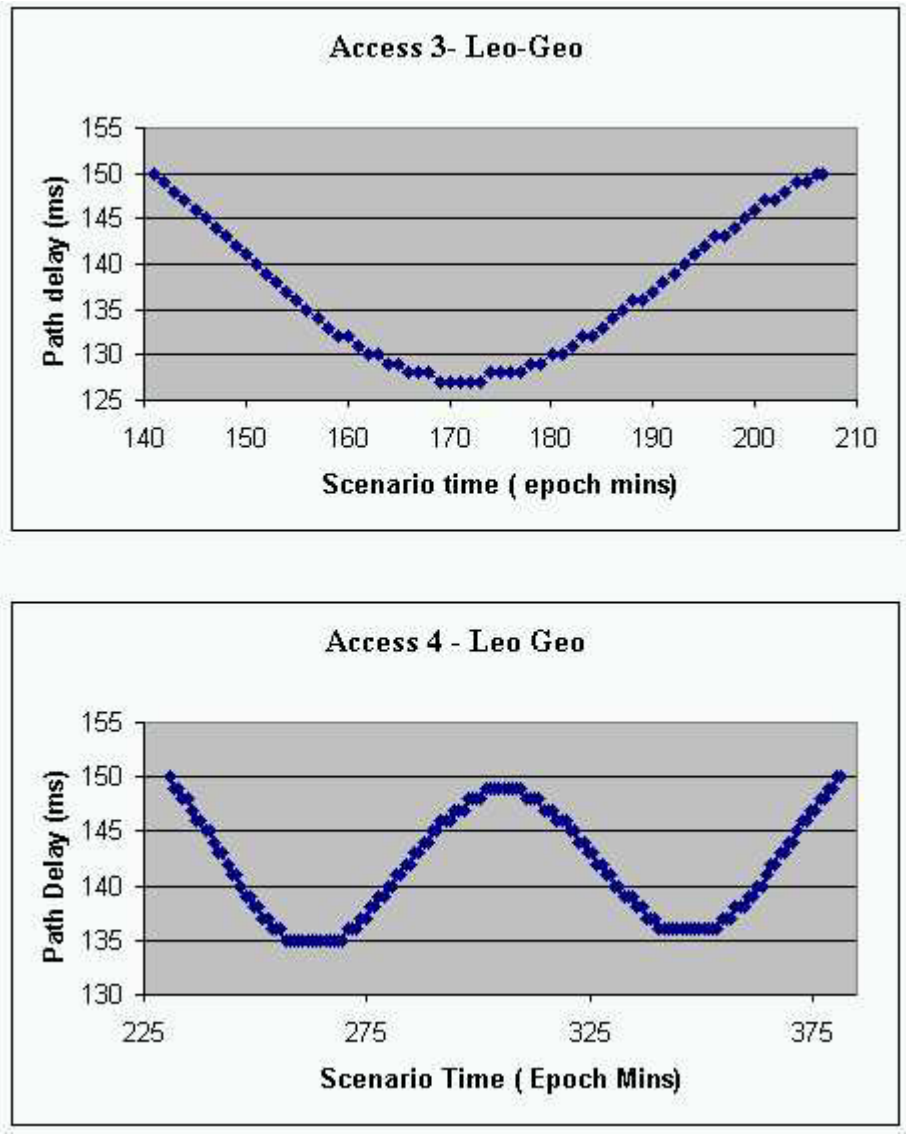


Fig 18: Propagation Delay versus Time for LEO-GEO Access

Observations:

- Minimum Delay = 112ms
- Maximum Delay = 150ms

- Maximum Delay variation per minute = 2 ms for access intervals around (0-70minutes), example Access 3.
- For access intervals in the range of 150 minutes (Access 4), the delay variations are uniform (1 ms per minute).

Conclusions:

- The transmission links for all the possible combinations are examined.
- The maximum delay variations per minute are 3 ms and minimum is 1 ms.

Based on these results, the algorithm for simulating the propagation delay is devised.

5.3 Algorithm for simulating Delay

5.3.1 Requirements

The algorithm is based on the requirements listed in Section 5.1. The propagation delay has to be simulated one-way and so the packets have to be delayed while they are sent on the physical device of the transmitting SBI node. The algorithm devised for simulating the delay is as follows:

- The packets have to be queued at the VETH layer and the queue size should be equal to the size of the in-flight bytes for that particular transmission link.

- So after the packets are delayed for the required amount of time, the packets have to be de-queued from the queue. The number of packets that have to be de-queued from the queue is equal to the number of packets that are in-flight on the transmission link.

- The number of packets to be de-queued at one time should be calculated. To compute this number, two parameters are required: transmission delay at the source and the propagation delay on the link.

- **Transmission Delay (seconds) = Average Packet Size (Bytes) / Bandwidth (bps)**

- **Total Number of packets in flight = Propagation Delay (milliseconds) / Transmission Delay (milliseconds)**

- The packet size can be a standard Ethernet MTU size of 1514 bytes. The Bandwidth is the link rate on the connection. The total number of packets multiplied by the average packet size gives the actual number of bytes that are in flight on the link.

- Another important parameter to be considered is the amount of delay variations on the link. This would change the queue size at the VETH layer. From the analysis results, the maximum delay variations are mostly 3ms per minute, which would increase or decrease the queue size by only 25 packets. The change in the queue size is also gradual.

- The above mentioned parameters and the delay value have to be passed to the VETH layer through a Delay Control Program. The Delay Control Program is a part of the Communications Controller Unit on the SBI Nodes. The control program receives the delay parameters from the Emulation Manager and accordingly sets the delay at the VETH layer through an *ioctl ( )* system call. The details of the control program are explained in the next section.

5.3.2 Calculations for the number of In-flight Packets

For the calculations, consider the maximum link speed on the SBI Node, which is 100Mbps. The average packet sizes can be 64, 1500 and 9180 bytes.

The first table calculated the transmission delay (in us) .

Packet Size (Bytes)	64	1500	9180
Transmission Delay (us)	5.12	120	734.4

Table 8: Transmission Delay for different packet sizes

To following tables give the calculations for the number of packets in flight for all the links considered in the analysis. The total number multiplied by the average packet size gives the total number of bytes on the link until the first packet reaches the destination.

1) LEO-Ground Station :

Delay Value (ms)	**Number of packets in flight for 3 packet sizes (bytes)**		
	64	**1500**	**9180**
Max Delay = 11ms	2148	92	15
Min Delay = 3 ms	586	25	4
Delay variations (+/- 2ms)	+/- 391	17	3

Table 9: Number of bytes in-flight on a LEO-Ground Station

2) LEO-MEO

Delay Value (ms)	**Number of packets in flight for 3 packet sizes (bytes)**		
	64	**1500**	**9180**
Max Delay = 34ms	6641	283	46
Min Delay = 7ms	1367	58	10
Delay variations (+/- 3ms)	+/- 586	25	4

Table 10: Number of bytes in-flight on a LEO-MEO Link

3) LEO-GEO

Delay Value (ms)	Number of packets in flight for 3 packet sizes (bytes)		
	64	**1500**	**9180**
Max Delay = 150ms	29267	1250	204
Min Delay = 112ms	21875	933	153
Delay variations (+/- 2ms)	+/- 391	17	3

Table 11: Number of bytes in-flight on a LEO-GEO Link

4) MEO-Ground Station

Delay Value (ms)	Number of packets in flight for 3 packet sizes (bytes)		
	64	**1500**	**9180**
Max Delay = 24ms	4688	200	33
Min Delay = 11ms	2148	92	15
Delay variations (+/- 1ms)	+/- 195	8	2

Table 12: Number of bytes in-flight on a MEO-Ground Station link

5) MEO-GEO

Delay Value (ms)	Number of packets in flight for 3 packet sizes (bytes)		
	64	**1500**	**9180**
Max Delay = 163ms	31836	1358	222
Min Delay = 112ms	21875	933	153
Delay variations (+/- 2ms)	+/- 391	17	3

Table 13: Number of bytes in-flight on a MEO-GEO Link

Delay Value (ms)	Number of packets in flight for 3 packet sizes (bytes)		
	64	**1500**	**9180**
Constant Delay = 128ms	25000	1067	174

Table 14: Number of bytes in-flight on a GEO-Ground Station link.

5.3.3 Algorithm Flow

- The propagation delay has to be introduced after the value of the link bandwidth is set to the Constant Bit Rate. The queuing discipline used is Token Bucket Filter (TBF) for limiting the rate on the link.

- The packets have to be queued at the VETH Layer for a time equal to the propagation delay value before sending it to the physical layer.

- Once the queued packets are delayed for the required amount of time, the packets have to be de-queued. The amount of bytes that should be de-queued at one time is equal to the number of bytes that are in-flight on the link.

5.4 Delay Control Program

Figure 13 refers to the SBI Node Controls for introducing the propagation delay on the VETH devices.

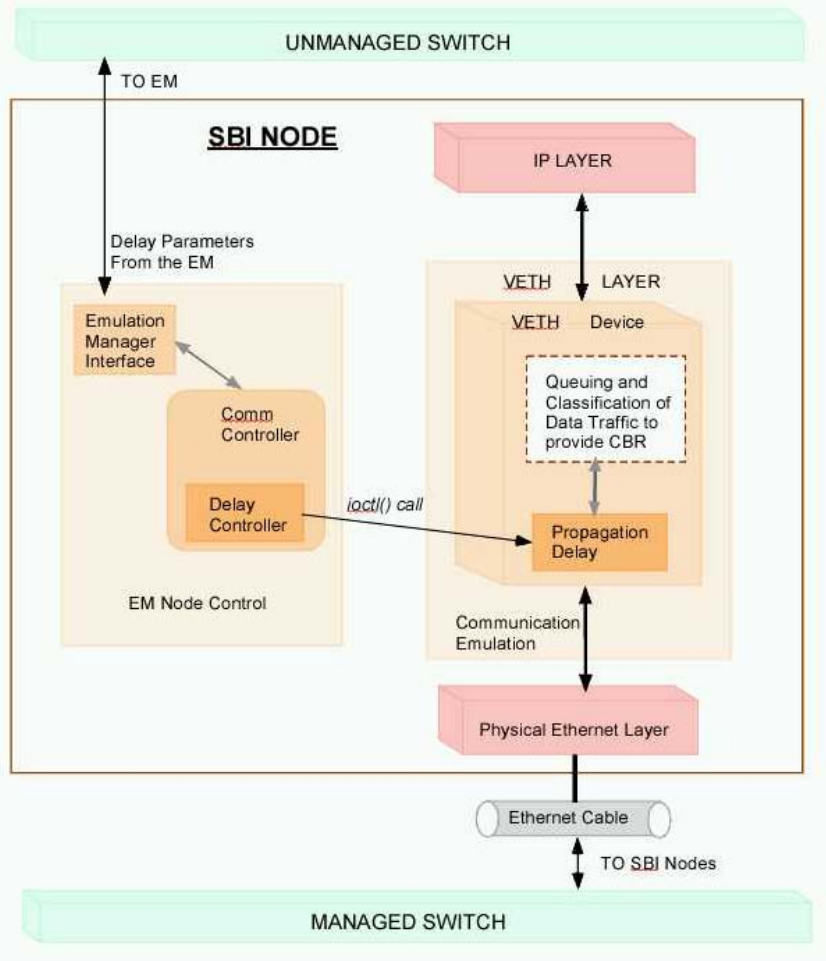


Fig 19: SBI Node Controls to simulate propagation delay

The Delay parameters are obtained from the Emulation Manager through the Manager Interface. These parameters are forwarded to the Delay Controller module in the Communication Controller Unit. The parameters passed to the control program from the Emulation Manager are:

- The virtual device on which the propagation delay has to be simulated.

- The link rate and the average packet size to calculate the transmission delay on the link.
- A complete STK report giving the Access times and the propagation delay values (in milliseconds) on the transmission link for the entire simulation time.

The STK report is in steps of seconds. The control program utilizes the "Access Report" to get the time duration (in seconds) for each access. Further, within each access the control program calculates ranges for which the delay remains constant. The control program puts the information in the following format:

1. Start time (seconds)
2. Stop time (seconds)
3. Total Duration (seconds)
4. Propagation Delay Value (milliseconds)

Getting the delay values from the calculated ranges, the values for transmission delay and the queue size (which represents the total in flight bytes) are calculated. These calculated values along the delay value is passes to the corresponding virtual device through the *ioctl( )* call. The control program keeps track of the time for which the delay remains constant through the calculated delay range by using *sleep( )* function. In case of any delay change, the parameters are passed to the VETH layer through the control program.

5.5 Implementation at the VETH layer

The implementation for the propagation delay involves extension to the functionality of the VETH layer. Some additional functions have to be added to the existing framework of the VETH layer. This section talks about the functions that have to be added to the VETH layer for simulating the propagation delay.

5.5.1 Additional data structures in the VETH Layer

The packets have to be queued to the VETH layer to simulate the delay. Each instance of the queue is represented by *struct queue_element*. The queue to store these instances is represented by *struct times_queue.* Every VETH device has a *times_queue* associated with it. The *times_queue* is an additional field in the *veth_device* structure for the VETH device. Program 5.1 details out *struct times_queue* and *struct queue_element*.

Program 5.1: Structures for queuing packets at the VETH layer

```
Struct times_queue{
      Struct queue_element *list; /* Elements of the queue */
      Struct timer_list wd_timer; /* Watchdog Timer */
      Long limit;                 /* Limit for the queue*/
      Struct packet_stats stats;  /* Statistics for the queue */
      Long inflight_bytes;        /* Total in-flight bytes
                                   * on the link */
      long delay;                 /* propagation delay on the
                                   * link */

};


struct queue_element{
      struct sk_buff *skb;       /* Packet Buffer */
      struct timeval time;       /* Timestamp the packet */
      long  delay_us;            /* Propagation delay for the
                                  * skb packet */
      struct queue_element *next;
      struct queue_element *prev;
      struct times_queue *Queue; /* Queue to which this element
                                  * is attached */
};

struct packet_stats{
      long bytes;                /* Number of "skb" bytes queued
                                  * in the times_queue */
      int packets;               /* Number of "skb" packets */
      int drops;                 /* Number of packets dropped */
};
```

The *struct times_queue* represents the queue to place the packets at the VETH layer before transmission. The individual elements of the queue are represented by *struct queue_element.* This structure stores the packet and the time stamp, when it was queued. It also stores the propagation delay associated with that packet.

The *times_queue* also has a watchdog timer incase the packets aren't de-queued at the right time. The *veth_device* structure has an entry to Times Queue that is associated with the device. This queue delays the packets to be sent on the physical device. It also stores the value of the propagation delay on the link. This is the value passed through the *ioctl( ) s*ystem call.

5.5.2 Additional Functions for the VETH layer

These functions can be added as an extension to the VETH layer framework. The packets are received from the IP layer to the VETH layer through *veth_send( )* function. The packets are enqueued on the Times Queue in this function. While de-queuing the packets, the timestamp on the packets is compared with the current timestamp. If the packets have been delayed for the required amount of time, then the packets are de-queued from the queue and the transmit function for the physical device is called.

Some additional functions have to be added to the VETH layer to implement the delay. These functions place the packets on the Times Queue and then de-queue it after adding the propagation delay. An additional option has to be added to the *veth_ioctl ( )* function to pass the delay parameters to the VETH layer.

Some of the additional functions added to the VETH layer:

- **Veth_delay_init:**

  This function is invoked from *veth_ioctl call ()* to assign the value of the delay parameters to the corresponding fields of *struct times_queue.* The data structures are stated in Program 5.1. Program 5.2 states the pseudo-code for initializing the fields of *struct times_queue.*

Program 5.2: Initializing the fields for queue limit and delay in Times Queue

```
times_queue->limit = inflight_bytes + TOLERANCE; /* from ioctl
                                                  *call */

times_queue->delay = prop_delay;  /* from ioctl call */
```

The fields *times_queue->limit* and *times_queue->delay* are passed from the Delay Controller Module through the *ioctl( )* call.

- **Veth_enqueue:**

  This function gets the packet from the IP layer. It creates an instance of struct *queue_element* to store the packet and the time stamp, when it was enqueued. This function also checks for the maximum queue limit for the Times Queue. In case the queue doesn't exceed the limit, the function *veth_enqueue_tail( )* is called, which queues the packet at the tail of the queue.

- **Veth_enqueue_tail:**

  This function queues the packet at the tail of the queue. Each queue element has a pointer to its previous and the next element in the Times Queue.

- **Veth_enqueue_head:**

  This function queues the packet at the head of the queue. This function is called from *veth_dequeue ( )* function. In case the packet is not delayed for the required amount of time, then the packet has to be placed at the head of the queue to be de-queued again.

- **Veth_enqueue_head_init:**

  This function initializes the head of the queue.

- **Veth_dequeue:**

  This function de-queues the packet from the Times Queue. It compares the time Stamp on the packet with the current time stamp. If the time difference equals to the propagation delay for the packet, then the packet is transmitted to the physical device. Otherwise, the packet has to be queued at the head of the Times Queue by calling *veth_enqueue_head ( ).*

- **Veth_dequeue_head:**

  This function de-queues the packets from the Times Queue. It is called from *veth_dequeue ( )* function to remove the packet from the queue.

- **Veth_watchdog:**

  This function is invoked when the watchdog timer related to the Times Queue expires. This timer expires if the *veth_dequeue ( )* function is not called for a certain time. This function de-queues the packet by calling *veth_dequeue_head ( )* and calling the transmission function for the hardware device.

In case, the packets are not transmitted successfully on the physical device, the *veth_requeue( )* function is called, which queues the packet again at the head of the queue.

5.5.3 Modifications to the existing VETH Layer Functions

- **Veth_ioctl**

  This function can take another option to pass the *delay parameters* to the VETH layer. The parameters passed are:

  - Propagation Delay (microseconds)
  - Total Number of bytes that are going to be in-flight on the link.

The parameters are passed from the control program into this function through the *ioctl()* call. These values are assigned to the fields of *struct times_queue* as stated in Program 5.2. The ioctl calls the *veth_delay_init ( )* for assigning the delay parameters to the corresponding fields of Times Queue.

- **Veth_init**

  This function is modified to initialize the fields of the data structures used for delay. The data structures are stated in Program 5.1. Program 5.3 states the pseudo-code for initializing the data structures for propagation delay.

Program 5.3: Initializing Data Structures for Delay in veth_init

```
veth_enqueue_head_init (vethdevice->times_queue);
times_queue->limit = 0;
times_queue->delay = 0;
init_timer(&times_queue->wd_timer);
times_queue->wd_timer.function = veth_watchdog;
times_queue->wd_timer.data = (unsigned long)times_queue;

times_queue->stats.bytes = 0;
times_queue->stats.packets = 0;
times_queue->stats.drops = 0;
```

- **Veth_destroy:**

  When the VETH device is destroyed, the memory occupied by the *struct times_queue* and *struct queue_element* has to be freed.

5.5.4 Function Flow for simulating the delay

The packets coming from the IP layer are subjected to CBR control and sent to the transmit function for the VETH device. The packets are placed in the queue and de-queued only after they are delayed for the required amount of time. This section explains the sequence of execution of different functions for sending the packet from the IP to the physical device layer.

5.5.4.1 Queuing Packets for Inserting the delay

The packets come to the VETH layer through the *veth_send ( )* function. In case the packets have to be delayed, the *veth_send( )* function calls the *veth_enqueue ( )* function to queue the packets on the Times Queue. In the queuing function, initially an instance of *struct queue_element* is created to store the packet. The packet is time stamped when it is queued. The other elements include the delay associated with the packet. This value represents the value of the Propagation delay for the VETH device.

The Times Queue has a limit, which is the sum of the in-flight bytes and some tolerance. When the packet has to be placed in the Times Queue, the queue limit is checked. If the limit exceeds the maximum specified value with the addition of the packet, then the "skb" packet is dropped or otherwise the packet is queued.

Once the queue element is created, it is en-queued at the tail of the Times Queue using *veth_enqueue_tail ( )* function. The Times Queue is attached to the corresponding VETH device. Each queue element also has a reference to the queue in which the elements are placed.

5.5.4.2 De-queuing the packets for transmission

The *veth_send ( )* function also calls the *de-queue* function of the packets. The de-queue function at the VETH layer, which is *veth_dequeue( ),* removes the packet from the queue. The packet is de-queued from the head of the queue by calling *veth_dequeue_head( )* function. The time stamp on the packet is compared with the

current time. If the difference between the two times equals the propagation delay value, then the packet has been delayed and can be transmitted on the physical device driver.

If the time difference doesn't equal the propagation delay value, then the packet is en-queued at the head of the Times Queue until it is delayed for the specified time using *veth_enqueue_head ( )* function.

Once the packet is queued, then the transmission function for the physical device is called. In case, there is an error in transmission of the packet on the physical device, the packet is re-queued on the Times Queue.

Chapter 6

6 Conclusions and Future Work

6.1 Conclusions

The SBI project aims at creating an Internet between the EOS satellites and incorporate routing and switching capabilities in them. The SBI software will enable the satellites to route their data to other satellites and grounds stations and not depend on the communication satellites such as TDRSS for relaying their information. The SBI emulation system described in this thesis will test the software on multiple scenarios. The proposed emulation system models an entire satellite system, which includes the emulation of the space and the ground segments and the communication between them. Emulation of communication links between these elements is a challenging task and its design involves careful considerations.

This thesis work presents a convincing design for emulating the satellite communication links. A satellite communication link has the following important features, which need to be considered during its design:

- A satellite transmission link is established between the instrument channels on the satellites and ground stations or other satellites. Each instrument has a separate communications channel (RF or Optical).
- The satellite link needs to have a dedicated bandwidth for the entire transmission duration to provide Constant bit rate (CBR) service to the signals.
- On account of large distances, the signals on these links suffer from high propagation delays and bit errors.

The design work presented in this thesis covers all these major aspects to emulate the communication on the satellite link. This thesis work makes the following contributions:

- It describes a design for emulating the communication channels through Virtual Ethernet (VETH) devices. The connection between two VETH devices emulates the communication link.
- It provides a convincing description for utilizing the Quality of Service algorithms in Linux to provide CBR service on the virtual Ethernet connections. This mechanism provides a dedicated bandwidth on the connection and emulates the CBR feature of the satellite transmission link.
- It presents an algorithm for simulating high propagation delays on the emulation link. The propagation delay values represent the actual path delays occurring on the satellite link.

6.2 Future Work

This thesis presents a design for emulating a simple channel to channel communication link. The work does not include the mechanism for modeling bit errors on the link. The prototype design for communication emulation, described in this thesis can be modified to model the actual *Bit Error Rate (BER)*. This could be achieved by introducing bit errors in the packets during transmission.

A convincing design has to be implemented to test its correctness. So one of the future tasks would be to implement the communication emulation unit as a separate application, test and evaluate it and then integrate it with the entire emulation system. The communication emulation unit can be tested on the emulation system hardware.

References

[1] Destination: Earth, The Official Website for NASA's Earth Science Enterprise, http://www.earth.nasa.gov/science/index.html

[2] NASA's Earth Observing System (EOS) Home Page, http://eospso.gsfc.nasa.gov

[3] Minden, G.J., Evans J.B., *Architecture for Space Based Internets,* NASA ESE Proposal, December 2000.

[4] Bleazard, G. B., *Introducing Satellite Communications,* NCC Publications, England, 1985.

[5] Inglis, Andrew F., Luther, Arch C., *Satellite Technology: An Introduction,* Focal Press Publications, U.S.A, Second Edition, 1997.

[6] TDRSS Online Information Center, http://nmsp.gsfc.nasa.gov/tdrss

[7] Searl, Leon S., *Space Based Internet System Architecture,* SBI Project, ITTC, University of Kansas, February 2001.

[8] Baliga, Sujit R., Rallapalli, Sandhya, *Space Based Internet Emulation Software Architecture Design,* SBI Project, ITTC, University of Kansas, March 2001.

[9] Roelof, Jonkman J. T., *Netspec: Philosophy, Design and Implementation,* MS Thesis, University of Kansas, February 1998.

[10] House, Sean B., Niehaus, Douglas, Sanchez, Ricardo, *Virtual Network Devices,* Technical Report, ITTC-FY00-TR-13200-X, August 2000.

[11] Almesberger, Werner, Linux Traffic Control- Implementation Overview, *ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz,* Technical Report SSC/1998/037, EPFL, November 1998.

[12] Radhakrishnan, Saravanan, Linux-Advanced Networking Overview, *http://qos.ittc.ukans.edu/howto.ps,* August 1999.

[13] Wagner, Kurt, Short Evaluation of Linux's Token-Bucket-Filter (TBF) Queuing Discipline, http://www.cosy.sbg.ac.at/~kwagner/tbf02_kw.ps

[14] Alexey Kuznetsov, iproute2 package, *ftp://ftp.inr.ac.ru/ip-routing/*

[15] iproute2 + tc notes, http://snafu.freedom.org/linux2.2/iproute-notes.html

[16] References on CBQ (Class-Based Queuing), http://www.aciri.org/floyd/cbq.html.

[17] Overview of Satellite Tool Kit, ITTC, University of Kansas, */projects/SBI/otherSW/stk/STKv4.1/STKData/Help/Overview.htm*

[18] Evans, J.B. , Minden, G.J., Prescott, G., Shanmugan, K.S., Frost, V.S., Petr, D. W., and Plumb, R., *Rapidly Deployable Radio Network",* IEEE Journal on Selected Areas in Communications, March 1999.

Appendix

Appendix A : Commands and Examples relating to VETH devices

- Creating a Virtual Device on device eth1 having a MAC Address 00:04:86:00:00:01

  Vethctl –c eth1 00:04:86:00:00:01

```
Vethctl: Virtual Device veth0 created successfully
```

- Listing the 3 virtual devices created on physical device eth1 along with all the details.

  Vethctl –l

```
Number of devices created: 3

List of devices:
  On eth1:  3virtual devices
Virtual device   Physical device  itfNum     Mac    Address
veth2            eth1            2     00:04:86:00:00:03
Veth1            eth1            1     00:04:86:00:00:02
Veth0            eth1            0     00:04:86:00:00:01
```

- Deleting a virtual device with interface number 0.

  Vethctl –d 0

```
ItfNum to be deleted: 0

Vethctl: Virtual Device veth0 destroyed successfully
```

- The virtual devices can be viewed through *ifconfig* command.

Ifconfig veth1

```
veth1     Link encap:Ethernet  HWaddr 00:04:86:00:00:02
          inet addr:10.67.2.11  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
```

ifconfig eth1

```
eth1      Link encap:Ethernet  HWaddr 00:80:C8:B9:02:40
          inet addr:10.67.2.1  Bcast:10.255.255.255  Mask:255.0.0.0
          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
          Interrupt:9 Base address:0xcc00
```

Appendix B: Script for setting up TBF and CBQ queuing disciplines

Consider Fig 10: for the scenario. Node A and Node B are observational Satellites having data rates at 450Kbps and 665 Kbps respectively. These nodes have a connection from their *veth1* devices to Node X on *veth1 and veth2* respectively. The *veth3* of Node X is connected to *veth3* of Node Y.

Nodes A and B have TBF queuing disciplines on their virtual devices. The TBF queues limit the rate on the link to the data rates specified.
IP addresses are set as follows:

Veth1 on Node A: 10.67.7.1

Veth1 on Node B: 10.67.9.1

Veth1 on Node X: 10.67.1.1

Veth2 on Node X: 10.67.1.2

Node X takes 2 traffic flows:

Node A (*veth1 – 10.67.7.1)* to Node C (*veth1 10.67.8.1)*

Node B (*veth1 – 10.67.9.1)* to Node D (*veth1 10.67.11.1)*

- Setting up a TBF queue having rate 450Kbps on *veth1* of Node A.

```
tc qdisc add dev veth1 root handle 10: tbf rate 450kbit burst
450k/8 limit 450k
```

- Setting up a TBF queue having rate 665Kbps on *veth1* of Node B.

```
tc qdisc add dev veth1 root handle 10: tbf rate 665Kbit burst
665k/8 limit 665k
```

- Node X has 2 traffic flows, each on *veth1 and veth2.* The destinations are Node C and Node D and so the traffic is transmitted on *veth3* to Node Y. Therefore, a

CBQ can be setup on *veth3* of Node X. The traffic can be classified to different classes on the basis of the source and the destination IP addresses. The individual classes can be allotted bandwidth as per the data rate specified.

```
# Setting up the root CBQ queue on veth3 of Node X. Bandwidth allocated to veth3 is 10Mbps.

tc qdisc add dev veth3 root handle 10: cbq bandwidth 10mbit avpkt \
1000 allot 1514 cell 8 mpu 64

#Attaching the root class to the CBQ queue and creating 2 classes within for the two traffic flows.

tc class add dev veth3 parent 10:0 classid 10:1 cbq bandwidth 10Mbit rate 10Mbit allot 1514 weight 1Mbit prio 8 maxburst 20 avpkt 1000

tc class add dev veth3 parent 10:1 classid 10:100 cbq bandwidth 10Mbit rate 450kbit allot 1514 weight 45kbit prio 2 maxburst 20 avpkt 1000 bounded

tc class add dev veth3 parent 10:1 classid 10:200 cbq bandwidth 10Mbit rate 665kbit allot 1514 weight 66kbit prio 5 maxburst 20 avpkt 1000 bounded


# Creating U32 classifiers to classify the packets to the classes. The filters match each flow on the source and destination IP address combination

tc filter add dev veth3 parent 10:0 protocol ip prio 100 u32 match ip dst 10.67.8.1 match ip src 10.67.7.1 flowid 10:100

tc filter add dev veth3 parent 10:0 protocol ip prio 200 u32 match ip dst 10.67.9.1 match ip src 10.67.11.1 flowid 10:200
```

APPENDIX C: Using STK for Delay Analysis

Appendix C describes the set of STK commands that can be given by the user to get the "Access" and "Delay" Reports for a particular transmission link.

- STK has a Connect module, which allows the user to setup a TCP connection with STK to obtain information about the scenario. The set of commands can be typed in a file and the executed through "*AgIPCExp"* command.

  ```
  AgIPCExp –f [socketName] < [filename]
  ```

  Referring to the scenario discussed in Section 5.2.2, the STK Connect module can be used to load the different elements of the scenario and obtain the "Access" and "Delay" Reports for various transmission links. An example of the script is shown below:

  ```
  # Load the Scenario "gnd_3sats"
  New / Scenario gnd_3sats

  # Set the Animation and the Simulation time for the scenario
  AllowAnimationUpdate Scenario/gnd_3sats ON
  Animate Scenario/gnd_3sats "1 Jan 2000 00:00:00.00" "2 Jan 2000 00:00:00.00"

  # Obtain an "Access" Report and a "AER" Report for a GEO-Ground Station transmission link in access_geo_gnd.txt and delay_geo_gnd.txt respectively. AER report gives delay versus time.
  ```

```
Report Scenario/gnd_3sats/Satellite/GEO SaveAs "Access"
"access_geo_gnd.txt"
Scenario/gnd_3sats/Facility/gnd_station

Report Scenario/gnd_3sats/Satellite/GEO SaveAs "AER"
"delay_geo_gnd.txt" Scenario/gnd_3sats/Facility/gnd_station

Unload / Scenario/gnd_3sats
```