Real-Time Networking for Quality of Service on TDM based Ethernet

by

Badri Prasad Subramanyan

B.E. (Computer Science and Engineering),
Bangalore University, Bangalore, India
September 2001

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

Dr. Douglas Niehaus, Chair

Dr. David Andrews, Member

Dr. Jerry James, Member

Date Thesis Accepted

Dedicated to my family and friends

Acknowledgments

I would like to thank Dr. Douglas Niehaus, my advisor and committee chair, for providing guidance during the work presented here. I would also like to thank Dr. David Andrews and Dr. Jerry James for serving as members of my thesis committee.

I would like to thank my family for their support and encouragement. I would also like to thank my sister and my close relatives who have been by my side when I needed them the most. I would also like to thank my younger sister for keeping me smiling.

I would also like to thank Hariprasad Sampathkumar for his collaborative work with me. I would also like to thank Tejasvi Aswathnarayana, Hariharan Subramanian, Deepti Mokkapati, Ashwinkumar Chimata and Noah Watkins for their help during the course of my thesis.

I am grateful to Danico Lee and Dr. Costas Tsatsoulis who gave me an opportunity to work as a Graduate Research Assistant and funded me during the course of my thesis.

I would like to thank my roommates who have made my stay in Lawrence memorable. I would also like to thank all my friends who have directly or indirectly helped me with my thesis work.

Abstract

The most commonly used LAN technology - the Ethernet, suffers from the fact that transmission of a message across the network is non-deterministic. There have been many suggestions provided like Time Division Multiplexing which make the network more deterministic. However, this deterministic network does not differentiate between processes and provides the same Quality of Service for all applications. Here we modify the Linux network stack to provide end-to-end Quality of Service for real-time applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A Local Area Network is a multi-access channel or a shared media where all the systems in the LAN share or access the same media to communicate with the other systems. A multi-access channel can use static channel or dynamic channel methods to allocate the channel for different systems in the LAN. Static allocation of the channel can be configured independent of the host systems, but dynamic allocation of the channel would need the systems in the LAN to communicate with each other to determine who gets to use the channel. The Medium Access Control (MAC) Layer, which is part of the Data Link Layer of the network stack, implements different protocols which are used to determine who goes next on a multi-access channel. The MAC layer implements different protocols depending on the hardware and the type of network. Some of the common LAN protocols are Ethernet or IEEE 802.3, Token Ring Protocol or IEEE 802.5 and Token Bus Protocol or IEEE 802.4. The Ethernet is the most widely used LAN technology due to its simple protocol design. Ethernet uses the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) technique to share the multi-access channel.

We discuss the CSMA/CD protocol in greater detail in Section 1.1. We discuss about the shortcoming of the present Ethernet in Section 1.2. We explain about some of the existing approaches and the problems they address in Section 1.3. Finally we talk about the proposed solution leveraging KURT-Linux capabilities in Section 1.4.

1.1 CSMA/CD Protocol

Ethernet uses the Carrier Sense Multiple Access / Collision Detection (CSMA/CD) access method to provide a shared media between computers. In this system, each computer senses the carrier or listens to the shared media to check if the network is being used by any other system. A system transmits only if the network is clear and is not being used by any other system. This takes care of the situation that no system uses the shared media when a single system has already acquired it and using it to transmit.

Ethernet uses Collision Detection (CD) in order to take care of situations when two or more systems try to transmit through a clear network at the same time. A collision is said to occur when two or more systems try to transmit a packet through the same media at the same time. CD detects a collision in a network after the computer transmits a packet, by listening to the network to check for any collisions to the transmitted packet. In the case of no collision, the transmission took place successfully. In the case of a collision, the computer stops the transmission immediately and transmits a 32-bit jam sequence instead. The jam sequence ensures that any other node, which may currently be receiving this frame, will receive the jam signal in place of the correct 32-bit MAC CRC, and discard the frame. The frames from all the systems involved in the collision need to be resent, as the frames were lost in the collision. So, each of the computer backs off for a random amount of time and retransmits the packet once again. This process continues until each of the packet is transmitted successfully. The amount of back off time is randomly selected based on the equation (r x 51.2 μs), where r is a random number between 0 to 2^{10}-1.

Ethernet has been the most widely accepted shared media access method for many reasons. Ethernet is one of the cheapest means of providing a shared media between computers. The set up cost of Ethernet would include setting up NIC cards and connectors on computers. Hubs, switches and cables are used to interconnect the computers. All these components are available at a reasonable price. Ethernet also gives the flexibility to operate at varying data rates; from 10Mbits/s to several Gigabits/s. Ethernet's wide acceptance has also brought wide availability of hardware and support. It is sim-

ple in terms of use, installation and configuration. The wide acceptance of Ethernet has made the Ethernet network adapters a standard feature within present day computer systems.

1.2 Shortcomings of Ethernet

Despite wide acceptance, Ethernet has a number of shortcomings. Ethernet was designed to provide a simple solution to using shared media. Typical Ethernet networks have a lot of collisions. Every collision and loss of packet is bounded by retransmission of the packet. Retransmissions brings randomness into network behavior, as each system involved in collision backs off for a random amount of time. This delay due to the collisions and retransmission can be overlooked in a normal scenario when the load on the network is low. However, when the load on the shared media is increased, then the message delay increases further due to a greater number of collisions. Real-time applications require predictable Quality of Service. Hence, this non-deterministic model of Ethernet, coupled with the increased delay, makes it unsuitable for using it for real-time networking purposes.

1.3 Existing Solutions

In order to make the Ethernet support real-time applications, we have to make Ethernet behavior deterministic. There are many suggestions that have been formulated over a period of time to solve this problem. The suggestions can be basically divided into two categories - hardware solutions and software solutions. Hardware solutions like SCRAMNet and Token Ring Protocol bring in real-time capabilities to the network by changing the hardware of the systems. Software solutions like RTnet and Rether try to achieve real-time capabilities by modifications to the software and no modifications to the hardware. The hardware solutions normally have better real-time capabilities, whereas the software solutions provide more economical solutions with broader applicability. The solutions have been discussed in greater detail in Section 2.

One of the solutions, to achieve real-time capabilities, is to implement Time Divi-

sion Multiplexing over Ethernet. As the name suggest, this method multiplexes different systems' use of the LAN based on time. In this method, each computer in the Local Area Network is allotted a time-slot, when it can transmit a packet. A computer would transmit only in the allocated time-slot. As a result of this, at any point of time, only one machine would transmit hence avoiding collisions. This would also make the network behavior more deterministic as there would be no collisions, hence avoiding back off or retransmissions. By this method, we are essentially splitting up the total bandwidth among all the machines in the LAN, and providing each computer with an assured bandwidth. The performance of this system may be worse than Ethernet when we have a single machine transmitting a large quantity of data due to the reason that the transmitting system cannot use the other time slots even if it is unused by the other systems. However, this method provides an assured bandwidth and the performance does not drop below the assured bandwidth even when all the systems in the network are transmitting a large quantity of data.

1.4 Proposed Solution

We consider building a system using Time Division Multiplexing over Ethernet to provide Quality of Service (QoS) that would be appropriate for real-time applications. Once the network is made deterministic, we need to study the Linux operating system, to look for other scenarios which can cause delays to the packet flow through the kernel. The Linux network stack is modified to reduce these delays that are present in the packet flow to achieve predictable end-to-end delays. Reducing these delays not only makes the packet flow through the kernel more deterministic, but also gives it real-time capabilities as it is more predictable. We also add other features which lets us differentiate between the real-time and non real-time packets, so that we can improve the performance of the real-time packets through the network stack.

The rest of the paper first discusses related work in Chapter 2, and then describes the background work in Chapter 3, explains the implementation of our system in Chapter 4. Chapter 5 describes the experiments performed along with the results

demonstrating the performance of our system and Chapter 6 presents the conclusion and the future work.

Chapter 2

Related Work

Ethernet was introduced in the 1970's by Xerox Corporation to provide a means of shared media network access. In due time people realized the shortcoming of Ethernet of not being able to satisfy the needs of a real time network. Research was carried out and various solutions were provided to solve this problem. The solutions can be classified broadly as hardware and software based solutions.

2.1 Hardware Solutions

Hardware solutions include those adding new hardware or modifying the present hardware to make the network appropriate for real-time applications. Most of the hardware modifications would require changing the NIC card to make the network appropriate for real-time applications. Hardware changes would also include minimal software changes to port software for the existing hardware. Some of the hardware solutions are shared media methods like SCRAMNet [6], RT-CRM [21], switched Ethernet and Token passing protocols like Token Ring [7] and FDDI[12]. Hardware solutions normally give better real-time performance, but are expensive to setup, as they require purchasing and setting up of new hardware devices.

2.1.1 Shared Memory Methods

Shared Memory or Reflective memory is one of the solutions to achieve real-time capabilities. In this method, each system in the network has a memory card which is used to store global data-structures of the system and the application. Each system can access its local copy of the global shared memory image. The NIC card and the network have been designed to maintain consistency among the content of the memories in all the systems in the network. We have discussed two methods which use the shared memory method to provide real-time capabilities. They are SCRAMNet and RT-CRM.

SCRAMNet

Shared Common Random Access Memory Network (SCRAMNet) [6] was designed first by Curtiss Weight Controls Embedded Computing Inc. This model uses a replicated shared memory model to provide low latency, system determinism, high throughput and guaranteed data delivery.

Each SCRAMNet card has a memory, which is available to the host computer like its physical memory. This memory is used by the computer to store its global data-structures. The network maintains consistency among the data stored in all the SCRAMNet cards in the computers. Whenever there is a change in a node's memory, the network protocol broadcasts this change and updates the memory on all the other nodes. The network uses a unidirectional ring topology to provide predictable data update latency. The speed of network transmission is much faster in this case as the broadcast does not have any CPU overhead and is solely a hardware communication link protocol.

SCRAMNet had different types of network cards available to work at varying network speeds. The improved version of SCRAMNet boards contain dual-ports, one of the ports is used by the system to read and write data into the memory. The other port is used to read and write data into the memory by the network. This dual-port architecture further improves the speed of the system.

Though SCRAMNet has proved quite effective in the field of real-time networking, it requires a hardware change, which imposes a cost for a network which is already up

and running. We may also need to port applications to work in this environment if the hardware changes of SCRAMNet are not encapsulated within existing APIs used by the application layer. This would also bring in the Inter-Process Communication code which would be used to encapsulate the shared memory model.

RT-CRM

Real-Time Channel-based Reflective Memory (RT-CRM) [21] uses the reflective memory technique to provide distributed real-time industrial monitoring and control applications. RT-CRM characterizes the requirements of a distributed industrial application and was designed to meet its needs. This method uses reflective memory to achieve this, but in a way different from that of SCRAMNet. SCRAMNet used a method where the hardware took care of reflecting each system's memory contents to every other system in the network. RT-CRM uses the idea that there is a data sharing pattern which is followed in distributed industrial applications where:

1. Data sharing is unidirectional
2. There are a set of producers/writers and consumers/readers, and each consumer would need data only from a subset of the producers. The number of producers are normally greater than the number of consumers.
3. Historical data from the recent past will be accessed very frequently.

The RT-CRM method is built upon the fact that the amount of memory on the NIC card is limited and needs to be used efficiently. Since all the consumers do not need the same data, using a SCRAMnet reflective memory may not be effective, as it would replicate all data on all nodes, thus consuming memory on many nodes to hold unused data. RT-CRM uses two key features to support a real-time distributed programming model.

1. Writer-push data reflection model.
2. Decoupling of writer's and reader's Quality of Service.

Data Reflection is accompished by Data Push Thread (DPA) residing in the writer's node. Every application in a distributed real-time monitoring and control system specifies the potential users of this data and also the frequency of update. This information could be used by the DPA thread to write the data into the consumer's memory. The consumer uses a couting semaphore to control the number of users who are accessing its memory at any given time. Each system has a set of QoS parameters which are available. The DPA thread updates the consumer's memory based on the QoS parameters from the producer system.

RT-CRM addresses the problem of real-time networking to a very specific domain of distributed real-time monitoring and control system. It also requires modification to the hardware and tends to be expensive in a small network scenario.

2.1.2 Switched Ethernet

A LAN, which uses a switch to connect the individual systems, is called a Switched Ethernet. Switches are replacing hubs these days due to their better performance, ease of availability and economical price. They are used in Ethernet to reduce the collision rate and improve the performance.

A hub basically repeats and transmits all the incoming frame out of all its ports. A switch is a smarter device which examines the header of the incoming packet to determine the MAC address of the destination device. A switch also has a built-in MAC-lookup table which shows the destination port for a given MAC address. Hence the switch transmits the incoming packet out on the appropriate port only. The switch thus provides a private collision domain for each of its ports and a guaranteed bandwidth per port. Switches are Layer 2 devices which drastically reduce the network congestions caused by using CSMA/CD medium access control protocol. Some switches also have inbuilt buffers which are used to store the incoming frames in situations where the out going port is busy. If several incoming packets are received for the same receiver in a short interval, they are queued in the buffer and send out one after another. But if the number of packets received are greater than the size of the buffer, then some packets are dropped.

A switch by itself does not ensure real-time behavior; it only improves the performance of Ethernet by reducing the number of collisions by reducing the congestion through communication path segmentation. Switched Ethernet offers simultaneous multiple transmission paths, provided the receivers in all these paths are different. Switched Ethernet has its own shortcomings, the main one being the latency at the switch. This latency varies, based on the switch and the number of frames transmitted to a particular segment on the switch. Switched Ethernet, though it improves the performance of Ethernet, is not deterministic in nature. It does not provide priority levels which distinguish the real-time and non real-time connections.

2.1.3 Token Passing Protocols

Token passing protocols are distributed polling protocols, where each machine is polled to check if there is any data to be transferred. The polling method used in these protocols is by passing a token among the systems. A system, which needs to transmit, needs to acquire a token before transmission. A token is a small control message, which is transmitted between systems. By limiting the number of tokens in the network to one, we can ensure that only one system transmits at a time thus avoiding collision in the network. There are many different token passing protocols implemented, most of them differing in the network topology and network connectivity. We talk about a few of the token based protocols in greater detail below.

Token Ring Protocol

Token Ring protocol or IEEE 802.5 [7], is a token based protocol which uses the ring topology. As previously mentioned, a token needs to be acquired by a system in order to transmit. The token, which is a small control message, is passed between machines sequentially from system to system. A system which needs to transmit, holds the token and transmits the data. The data, which is transmitted, moves from system to system in the ring. Each system repeats the data on the network, checks for errors, and makes a local copy of it, provided the data is destined for this system. When the data reaches the transmiting system, it acts like an acknowledgement for the reception of data. The

transmitting system removes the data; transmits more data if it has any, otherwise it passes the token to the next system. The Token Ring protocol can also support priority by setting some priority bits in the Token Ring frame. This protocol also uses some timers to prevent a single system from holding the token for a long time. One of the systems in the ring acts as the ringleader and is responsible to preserve the token. It takes care of a lost or corrupted token situation.

The Token Ring protocol has a deterministic response time and has a single system transmitting at any point of time; hence it avoids collisions due to multiple systems transmitting at the same time. The performance of the network does not drop with the increase in load. On the other hand, if a single transmission line in the ring fails, then the whole network would stop functioning because of an incomplete ring. The token ring protocol uses a specific Network Interface Card (NIC), different from the one used by Ethernet, to support it. This protocol also has the overhead of maintaining and passing the token. Due to this overhead, the throughput of Token Ring protocol is worse than Ethernet when the traffic is low.

Token Bus Protocol

The Token Bus protocol or the IEEE 802.4 [24], is a token based protocol which uses the bus topology. All the systems in the network are connected to the bus which has a master controller. A logical ring is formed among the systems, with the help of the master controller. The packets are transferred from one system to the next system in the logical ring, thus maintaining an order and direction of data flow. As previously mentioned, a token is passed between machines, which needs to be acquired by a system to transfer data. It is a collision free network as there is only one token at any point of time, thus guaranteeing that only one system transmits packets at any given point of time. The Token Bus protocol can also support priority using priority bits, and uses timers to avoid a system from overusing the network bandwidth. The Token Bus protocol is quite similar to Token Ring protocol except for two major differences; the different topology used by the two networks and the master controller present in the token bus protocol to create the logical ring.

The Token Bus protocol also has all the advantages of the Token Ring protocol. The bus structure also sorts the problem of a single problematic node or link bringing down the network. All these features are achieved by making the protocol more complex and adding more overhead to the network. The Token Bus protocol uses a specific NIC, different from that of Ethernet, to support it.

Fiber Distributed Data Interface (FDDI)

FDDI [12] is an improvement over the Token Ring protocol. FDDI is similar to Token ring protocol except that it uses two network connections between each system in the ring - one for each direction. This is designed so that the network can work properly even under a broken ring condition. The token is passed simultaneously on the network's inner and outer ring which back up each other. In the case of a station malfunction or a broken ring, the closest station closes the network loop by sending the token received in the inner ring to the outer ring and outer ring to the inner ring. This would remove the faulty system or connection from the FDDI ring structure, but would let the remaining systems function normally.

This has all the advantages of the Token ring network with more fault tolerance. FDDI uses optical fiber transmission links operating at 100 Mbps. This makes the network faster, but is expensive to implement in a LAN. Hence FDDI is mostly used as a backbone network. The throughput of FDDI is low, as at any time only one of the network connectors are used to communicate while the other one remains idle.

2.2 Software Solutions

Software solutions include only software modification without any changes to the hardware. Software solutions, in general are cheaper alternatives to achieve real-time capabilities, as they do not require any hardware changes. Some of the software solutions include RTNet, Rether, Traffic shaping and Master/Slave protocols. These have been expained in greater detail below.

2.2.1 RTNet

RTnet [15] is a hard real-time network protocol stack for RTAI. RTnet uses a software protocol, rather than modifying or adding new hardware to the computer system, to achieve bounded transmission delays.

RTnet has implemented the UDP/IP protocol stack including basic ICMP and ARP. RTnet uses the Linux network stack's implementation for TCP/IP. TCP represents the non real-time traffic generated by the computer. RTnet implements a protocol - Rtmac, to enable the flow of both, real-time and non real-time traffic on the computer. RTnet has a real-time enabled network adapter driver which controls the NIC card. The Real Time Media Access Protocol (Rtmac protocol) runs on top of this driver. There are 2 stacks built over this protocol, the RTNet's real-time protocol stack which implements UDP/IP and the Linux network stack which implements TCP/IP for non real-time traffic. The Rtmac implements a Virtual Network Interface Card (VNIC) device driver which emulates a Ethernet device driver for the Linux stack. So the implementation of the Rtmac layer is concealed from the Linux network stack. The Rtmac layer is implemented as an optional module which can be disabled if real-time networking is not desired or is already available.

The RTMac implementation also includes other features such as a basic TDMA protocol. In TDMA, every machine on the LAN has a fixed unique time slot when it is allowed to transmit. One of the stations is configured to be a Master and the rest of the systems act as slaves. The master is responsible for sending configuration and transmission slot information to the slave machines. The master has a global time schedule, based on which it sends synchronization packets at the beginning of each time cycle. The slaves use this synchronization signal to confirm their slot time. The master also distributes the global timestamp to the slave machines.

RTMac also provides a prioritized queue for the outgoing packets. RTMac implements 32 levels of outgoing queues, with the lowest one being reserved for non real-time protocols.

Another important feature in RTnet is the buffer management. A communication system might lock up when it sends or receives large amount of data, due to avail-

ability of buffers. Dynamic allocation of input and output buffer may not be sufficient because of the strictly bounded execution time of real-time processes. Hence RTnet implements a multiple pool based allocation mechanism for its fixed-sized buffers. It has a pool for each critical component, like NIC receive pools, user socket pools and VNIC pools.

RTnet gives a good software solution to the real-time networking problem. RTnet does not implement TCP, which forces the usage of TCP as a non real-time protocol. RTnet also re-implements network stack for UDP/IP, ICMP protocols, which would mean that at any given point of time there would be 2 network stacks loaded in the kernel which deviates from the idea of having a single Linux network stack. RTMac layer acts like a hardware abstraction layer between the Ethernet card and the Linux network stack. The Ethernet drivers need to be ported to the Rtmac layer to make them real time enabled adaptive drivers.

2.2.2 Rether

Rether [5] stands for Real-Time Ethernet protocol. Rether is mainly implemented for supporting smooth delivery of multimedia streams over the network. Distributed multimedia applications are time critical packets and cannot be used on Ethernet because of its non-deterministic nature.

The Rether project is implemented in software without any change to the Ethernet hardware of the computer. Rether is based on a token passing scheme that regulates the access to the network by passing a control token among the nodes of the Ethernet segment. A system needs to obtain and hold a token in order to transmit data. By reducing the number of tokens to one, we can see to it that only a single system transmits at any given point of time, thus bringing the collisions to zero. Rether has also implemented a hybrid mode of operation. In this mode, the setup can automatically switch between the Rether mode (Token passing mode to support real-time applications) and the CSMA/CD (Ethernet) mode depending on if there are any real-time connections active at that point of time. Rether also takes care of allocating some bandwidth for non real-time applications so that the non real-time applications don't starve for network

bandwidth in the Rether mode. The amount of bandwidth allocation for the non real-time applications is set such that the higher-level protocols using TCP don't timeout. Rether is already implemented and tested for 10Mbps and 100Mbps Ethernets.

Rether is a token based protocol with minimal changes to software and no changes to the hardware. Most of the changes to the software are to the lower layers and are encapsulated from the upper layers of the Linux protocol stack. Hence all applications on the system could run without even being changed or being aware of the existence of the changes for Rether. Rether is specifically designed for multimedia traffic on Ethernet, which include data transfers that occur periodically. Rether does not handle the situation of large bursts of data transfer between machines. If multiple machines try to transfer files at the same time, it can lead to contention for the token.

2.2.3 Traffic Shaping

Traffic shaping [20] builds a relationship between bus utilization and collision probability. Traffic shaping technique addresses the problem of delay in real-time packet transmission in a network where real-time and non real-time packets are concurrently transmitted. It stresses the fact that the delay in real-time packet transmission is due to two main reasons:

1. Contention of the real-time packet with the non real-time packet at the local node where they originate.

2. Collision of real-time and non real-time packets from different nodes.

Traffic shaping technique is built on the assumption that by keeping the bus utilization below a threshold would result in a desired collision probability. In order to resolve the problem of delay in real-time traffic, this technique uses two adaptive traffic smoothers - one at the kernel level and the other at the user-level. The kernel-level traffic smoother is installed between the IP layer and the Ethernet MAC Layer. This smoother takes care of prioritizing the real-time packet and avoiding contention between the real-time and non real-time packets on a local node. The user-level traffic smoother is built on top of the transport layer. This smoother is used to control the non

real-time traffic generated by a node. This traffic smoother controls the transmission of the non real-time traffic on a node to keep the network load below the threshold level. This would regulate the non real-time traffic generation to adapt itself to underlying network load conditions. This should reduce the collision of the non real-time traffic from this node with the real-time traffic generated on other nodes.

Traffic shaping is implemented on Linux kernel with minimal changes to the kernel and without any modification to the network protocols. Traffic shaping can be used only for soft real-time applications. It does not make complete use of the network bandwidth as it tries to keep the load on the network below a threshold which tends to drop the network throughput. It may not be able to satisfy the needs of the nodes when the network usage of all the nodes increases.

2.2.4 Master/Slave Protocol

The Master/Slave protocol tries to achieve determinism by using a centralized traffic controller called the master. Every other node present it the network transmit message on receiving an explicit control message from the master.

The Flexible Time Triggered (FTT) Ethernet protocol [17] is one such protocol, which uses the Master/Slave configuration to support hard real-time communication in a flexible and bandwidth efficient way. The key concept of this protocol is the 'Elementary Cycle', which are fixed duration time slots. The bus time is organized as an infinite succession of ECs, each of which are started by the master by sending a trigger message. The EC mainly consists of two windows - the Synchronous window and the Asynchronous window. The Synchronous window is subject to admission control and used for real-time traffic. The Asynchronous window on the other hand, is used for event-triggered communication.

The master node in this protocol plays the role of a system coordinator. The master maintains a database holding system configurations and communication requirements, and builds the ECs accordingly. The nodes on the other hand maintain a table identifying the synchronous messages it produces. Upon reception of an EC trigger, the slave node decodes the EC message to identify the synchronous message it needs

to transmit and queues it for transmission in the synchronous window. Though this protocol can maintain precise timeliness, they have considerable amount of protocol overhead. The Master/Slave model uses centralized control, which implies a single point of failure. This protocol needs to address the issues such as fault tolerance when the master node goes down. It may need a backup master or a method of electing a new master when the master node goes down.

Chapter 3

Background

The Linux system programming involves adding enhancement to the already available open source Linux code. In order to make this possible, we need to understand the working of the Linux operating system. The modifications needed to the present system may not be large depending on the type of enhancement required. However, in order to know the exact place and exact code which needs to be added to bring about the enhancement without breaking the rest of the system can be significant task. Most of the time spent in the project is on understanding the present system rather than modifying it. It is very important to understand the system well before a programmer begins to modify it, as half knowledge can cause more harm than good. A lot of time was spent studying the Linux network stack in order to identify the best and least intrusive way to bring real-time capabilities to the network protocol stack. All the background work done for the project has been summarized in this section. This section also includes the other tools and features used to understand and implement real-time networking for Quality of Service over Time Division Multiplexed Ethernet.

3.1 UTime

Linux uses a timer chip to maintain a sense of time. The timer chip usually provides a periodic interrupt where the period corresponds to 10ms in Linux 2.4.x and 1ms in Linux 2.6. This amount of timer resolution may not be sufficient for real-time require-

ments when we want to process an event at a particular time.

The Utime project [13] was implemented to improve the temporal resolution of Linux. In order to improve the resolution of the timer, the obvious suggestion would be to increase the rate at which the timer interrupts the system so that the timer chip would interrupt the kernel at a higher frequency. This would not be an acceptable solution as this would increase the overhead of running the timer service routines.

Utime handles the problem in a different way. It changes the mode of the timer from periodic to a one-shot mode. The one-shot mode allows the timer chip to interrupt the kernel at specified times rather than at periodic times. We would then use this method to interrupt the kernel at times when we need to run some computation rather than only at periodic intervals. The disadvantage of this method is that the timer chip needs to be programmed for each interrupt.

The Utime project also includes the addition of the Utime timerlist to the kernel. A timer contains a top-half which is the Interrupt service routine which is called when the timer expires, and a bottom-half which is the function added by the user which needs to be executed when the timer expires. The Utime timers are similar to the kernel timer, the only difference being the context in which the timer bottom half is executed.

When a kernel timer, which is added to kernel timerlist, expires, the top-half of the timer is executed as an Interrupt Service Routine. A flag is set to execute the bottom-half in the softirq context. We can notice that the time delay between the time when the timer expires and the execution of the bottom-half can be quite large depending on the number of interrupts that arrive at the system in that interval. Utime timers takes care of this by executing the bottom-half of the timer just after executing the top-half of the timer, in the same context without any delay.

Utime timers were used in this project to get accurate control over time. Utime timers were used in Time Division Multiplexing to enforce use of the Ethernet by a machine only during its assigned slot. Utime timers were critical at this point as the accuracy of these timers have a fundamental influence on the accuracy and efficiency of Time Division Multiplexing for the Ethernet resource.

3.2 DSKI/DSUI

Debugging the kernel code is much more complex than debugging user level code. The main reason for this is that the kernel is much bigger and complex code than most of the user level application. The other main reason being that there are no easy debugging techniques that can be used with the kernel as it can be done in the user space. The third reason being the large amount of concurrency that exists in the kernel. One of the simplest debugging technique which is used in the user space is the use of a print statement which log events on the standard output or into an output file. This method cannot be directly applied to the kernel as this would mean recompiling the kernel every time we include a new print statement. The kernel does not give us the flexibility to open a file and write out the log into it and the print statement also has a high overhead on the system.

The Data Stream Kernel Interface (DSKI) [4] is a pseudo device driver which lets us log events occuring in the kernel. This gathers performance data from the operating system and outputs it to the user in a presentable format. It also contains a time-stamp for each of the events.

The kernel code is modified to include instrumentation code which act as log points. Instrumentation points are defined as points which log an event when a thread of execution passes through this point in the kernel code. Each of these instrumentation points can be an 'Event', 'Counter' or a 'Histogram' depending on the kinds of data required. Once the user decides on the relevant code which needs to be studied, he can use the DSKI's external interface to collect events which are relevant to the code under study and the experiment being run. This external interface also takes other inputs including the events which need to be logged and duration of logging events.

The collection of events is in binary format to reduce the data collection overhead to minimum. Once the log file is collected in binary format, we can use various post-processing tools to get the event log in one of the more user-friendly formats. Postprocessing also includes a set of filters which could process only a subset of the event log based on the values set for the filters. DSKI not only provides an easy way of debugging the kernel, but also lets us change the desired points for instrumentation without

recompiling the kernel. The postprocessing also provides a GUI which can be used to visualize the occurrence of events on a timeline which would give a better perception of the events.

The Data Stream User Interface (DSUI) [4] is a similar instumentation method which is available for user level programs. A user needs to instrument the user level applications with the DSUI instrumentation points. Once this is done, these instrumentation points can be used to log events as with the DSKI. This is very useful when we need to check the user-level code along with a few kernel level instrumentation points using the same timeline. The DSKI and DSUI generate seperate data files, which can be combined into a single log file, based on the time of occurrence of the events, using different post processing routines. Once the log file is generated, it would containi, both the DSKI and DSUI instrumentations and this could be used to check the performance of the user routine with the kernel's timeline.

The DSKI was used in this project to understand the flow of control in the Linux network stack. The Linux network stack was instrumented under DSKI using different event tag values to get information regarding the flow of control through the network stack for different protocols. The DSUI was mainly used to compare the flow of control through the network stack and the flow of control through the user routine on the same timeline. DSUI and DSKI instrumentation points were also helpful in calculating the delay of packet processing as control and packets moved from the kernel space to the user space.

3.3 NetSpec

Netspec [16] [14] was a tool which was designed for network experimentation and testing. Netspec provides a way to run distributed applications from a single central location. Netspec provides a framework which enables a user to centrally control daemons running on other machines.

Netspec daemon is run on each of the daemon machines. The Netspec daemon also takes a configuration file which gives the path to different executables on the daemon

system. It also takes a few command line options which configures the daemon being controlled. The Netspec server routines takes an input file which contains information regarding the number of daemons, the mode in which each daemon needs to be run and the applications which each of the daemon needs to fork. Netspec also gives the flexibility to transfer files between the server and the daemon systems. Netspec can create and pass the configuration file to the daemon routine which configures the daemon for its assigned role. Once the tests are complete on the daemon system, the output file can be transferred back to the server routine. Using Netspec we can have a single point of control with easy reproductivity of the experiment.

Netspec also gives the server control of the daemon routine which is running on the remote system. The remote daemon is divided into four major routines - setup command, open command, run command and finish command. The Netspec server is given the flexibility to change the order and time of occurrence of these commands on the individual daemon systems.

Netspec is also given control of the time and order of execution of each of the daemon with respect to the other daemons. The Netspec server can run the daemon routines serially or in parallel. In the serial mode of execution, the execution of one daemon is completed before the execution of the other daemon is started. In the parallel mode of execution, the server calls and executes the daemons concurrently, i.e. each phase of each of the daemon executes concurently.

Netspec gives the needed flexibility to run distributed applications. Netspec was used in this project to control different tests in a Local Area Network to test the system and evaluate the performance of our approach to end-to-end Quality of Service over TDM based Ethernet. Netspec was used to run different programs and collect log information on different systems in a LAN. Once this was done, all the reports were returned to the server routine for analysis. Netspec not only gave a central location of control for the distributed experiments, but also a single location for storing and analyzing the results.

3.4 Group Scheduling Framework

The traditional Linux scheduler is a priority-based scheduler. Each process that runs on the system has a static and dynamic priority assigned to it. Every time the Linux scheduler needs to select a new process for execution, it calculates the goodness value of a process using the static and dynamic priority, and the process with the maximum goodness is selected to run next. In order to support real-time processes, Linux scheduler has 'rt_priority' which is used during the goodness calculation. This method works well when there is just one real-time process. When there are more than one real-time processes, they often compete with each other for the processor time. The traditional Linux Scheduler does not give direct control to the programmer over process execution.

The Group Scheduling framework [11] was built to overcome the shortcomings of the Linux scheduler. This model allows us to configure the scheduling semantics for selected Linux processes. The Group Scheduling framework treats every process, which needs the processor time, as a computational component. It gives the flexibility to arrange the control algorithms for these components in a hierarchical fashion where the hierarchic decision tree decides what to execute next. It also supports explicit control of computational components in the OS. When we talk about the processes, which use the CPU time, we can segregate them into 3 main types. They are:

Hardirq: Hardirqs are basically the hardware interrupt's service routines. These are also known as the top-halves. Whenever a hardware device needs processing, it raises a hardirq. It has the highest priority among the three types of computational components. Most of the I/O devices have an interrupt associated with them, which is called when the device needs the CPU time. Every interrupt has an Interrupt Service Routine (ISR), which is called when the interrupt occurs to process the interrupt. Each interrupt has a priority of its own. If an interrupt with a higher priority than the one being processed, occurs, then it is immediately sent to the CPU for processing, otherwise it is queued for later processing.

Softirq: Softirqs are similar to interrupts, but these are processes, which can be deferred for some time. The interrupt service routines need to be executed as soon as the interrupt occurs in a system. But executing the whole ISR might be time consuming, so the ISR is split into 2 parts - top halves , which need to be processed immediately and bottom halves, which can be delayed for a sometime. These bottom halves are normally processed as softirqs. These are processed when there are no more interrupts to be processed. Linux checks to see whether any softirqs were raised in different parts of the code and processes them at the earliest possible time. The kernel maintains softirq flags which are data structures that track the pending softirqs. Softirqs are scheduled using the `do_softirq` routine which checks for each of the softirq flags and processes the one, which is pending. This `do_softirq` routine is called from five different places in the kernel. They are:

1. When the `do_IRQ [arch/i386/kernel/irq.c]` routine finishes handling a hardirq, then it checks to see if there are any pending softirqs to execute.
2. When the local bottom half enable routine is called which re-enables the softirq.
3. When the `smp_apic_timer_interrupt [arch/i386/kernel/apic.c]` function completes handling a timer interrupt, then checks to see if there are any pending softirqs.
4. When the `Ksoftirq_CPU` thread runs which is started by the `wakeup_softirq [kernel/softirq.c]`.
5. When a packet is received at the network interface and the ISR for a packet reception is completed - this is applicable only to some of the drivers.

There are 4 basic softirq listed below based on their priority:

1. HI_SOFIRQ - This softirq processes the bottom halves of high priority interrupts.
2. NET_TX_SOFTIRQ - This softirq is responsible for transmitting a packet out of system.

3. NET_RX_SOFTIRQ - This softirq is responsible for processing a packet received by the system.

4. TASKLET_SOFTIRQ - This softirq processes the bottom halves for lower priority interrupts.

Process: Process is a generic term that is used for any instance of a program of execution. In our context we would define 'process' as any process other than a softirq or a hardirq. These processes also include the user level applications. When we talk about processes, the kernel level process always has a greater priority than the user level processes. The priority of the user process can be changed to execute the process faster and more often.

The Group Scheduling framework is made up of three main components - group, member and the scheduler. These three components are used to build up the group hierarchy. We explain these components in detail.

Group: A group is a special entity which forms a place holder for other entities. The group decides the internal structure of the Group Scheduling framework. A group contains members which are called as the group members. A group is also associated with a scheduler which decides the scheduling policy within the group. The scheduler selects one of the members based on its scheduling policy for execution. Each group has a unique number called the GroupID. A group is also associated with a unique name called the group name.

Member: A member is a member of a group. A member can be a computational component like a process, hardirq, softirq or another group. A group contains a list of members whose selection for execution is under its control. The scheduler selects one of the processes to be processed based on the scheduling function. By adding a group as a member of another group we can build hierarchical structure based on which processing takes place.

Scheduler: A scheduler is a decision making routine which uses some algorithm to select one of the group members for processing. There are many built in schedulers like the sequential, round robin and priority schedulers that can be associated with a group to schedule members. A group can be associated with any one of the built-in schedulers to achieve the desired scheduling discipline.

By using these features, we can build a hierarchy of group structure which can execute computations on the machine according to the policies we desire. The framework also gives the flexibility of having more than one group scheduling hierarchy configured, so that the hierarchy can be changed dynamically on runtime without restarting the system. There are many APIs available to the user which lets us modify the group scheduling hierarchy after the system is up without restarting the system. The Top group is the top most group in the group scheduling framework. The reference to this group is provided in the group scheduling framework. It also has routines which can be used to set any group as the top group of the framework.

The Group Scheduling framework also supports the concept of the programming model. A model is a loadable module which establishes the scheduling hierarchy. The model has the option to have its own hardirq and softirq processing routines which can be plugged in the place of the default routine using function pointer hooks. A model can have one or more Group Scheduling System Scheduling Decision Function (SSDF) structures which are established when the model is loaded. The model also defines the way in which this model executes each of the computational components. The model also has routines which lets the model select/deselect a computational component returned by the group scheduling hierarchy for execution. These routines give the model total flexibility to control execution as desired. We design a particular model to achieve a desired functionality and implement it using the scheduling hierarchy of the model.

The group scheduling framework has a few built in models which can be loaded on startup. The Basic model controls the system by default. We also have other models like the Vanilla Linux softirq model which emulates the working of the Vanilla Linux kernel, after gaining control over softirqs, using the Group Scheduling framework. We also have an option to select the TDM model which configures the Group Scheduling

hierarchy so that the kernel can be configured to perform Time Division Multiplexing on Ethernet.

We talk in detail about the TDM model as the Real-time networking model is built on the TDM model. The group structure of the TDM model is given in Figure 3.1. This hierarchy contains 3 groups. The 'Top' group is a sequential scheduler with 2 groups attached to it, a 'TDM' group and a 'Softirq' group. The first member to be executed by the 'Top' group is the TIMER_BH in order to get good timer interrupt responses. The TDM group has a TDM scheduler which executes the transmit softirq depending on the TDM schedule of the system. If the system is not in TDM mode, then this group is not used.

The Softirq group has a sequential scheduler. It has 5 members, 4 of which are the Linux softirqs which are scheduled in sequential order based on their priority. In the TDM model, the timeslot for transmission is very precious and needs to be used efficiently for transmit only. In Linux, the transmit softirq not only transmits a packet but also frees the packet's memory once the transmission is complete. Hence, under the TDM model, the Linux transmit softirq has been broken into 2 separate softirqs - one to transmit the packet and the other to free an already transmitted packet. This is done so that the scheduler only transmits during the TDM timeslot and does not spend any processing time in freeing the sent packets which can be done during any other time. The 5th softirq which is called the NET_KFREE_SOFTIRQ is used to free the memory of an already transmitted packet. This has the lowest priority and is the last in the list of members in the softirq group.

3.5 Linux Traffic Control

Linux provides a very rich set of tools for managing and manipulating the transmission of packets. These tools include a set of queuing structures which can queue and transmit packets. These tools are collectively called Linux Traffic Control [2]. This provides features which help provide Quality of Service on Linux.

The Figure 3.2 shows the method in which the kernel processes the incoming pack-

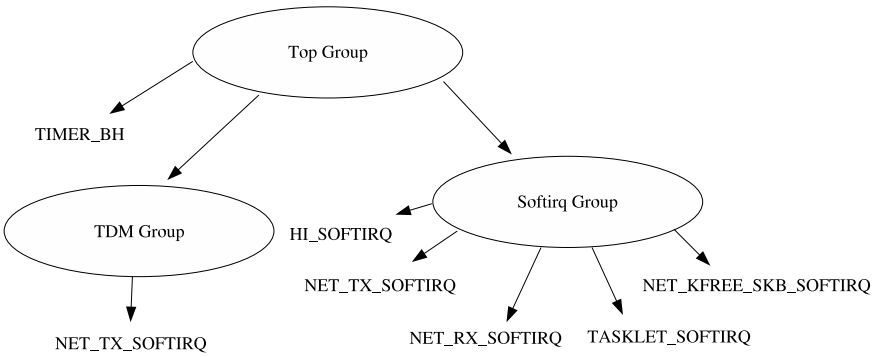

Figure 3.1: Group Scheduling Framework to implement TDM

ets and the locally generated packets on Linux. The input de-multiplexer checks every packet that comes into the system. If the packet is for the local node, then it is sent up the network stack, else it is sent to the forwarding routine which uses the routing table to lookup the next-hop for the packet. Similarly, locally generated packet comes down the network stack to reach the forwarding routine which uses the routing table to lookup the next-hop for the packet. Once this is done, the packet is queued to be transmitted on the output interface. This queue corresponding to the output interface forms the part of the Traffic Control. Traffic Control gives flexibility to build a complex combination of queuing disciplines and filters to control the packet flow through the output interface.

The three main components of the Linux Traffic Control are:

- Queuing Discipline
- Classes
- Filters

Each of these have been described in greater detail below.

Queuing Discipline: A queuing discipline includes a queue which is used to hold packets. Each queuing discipline is associated with an algorithm which controls the

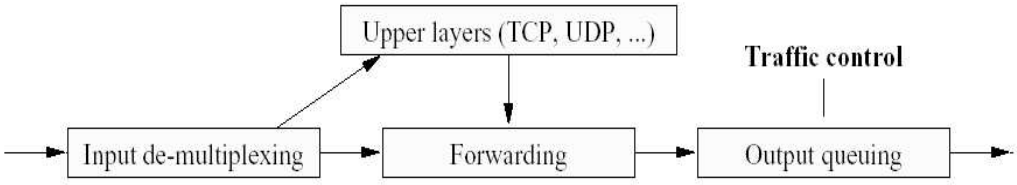

Figure 3.2: Packet processing on Linux

way in which the enqueued packets are treated. A simple queuing discipline is a FIFO (First-In First-Out) queue where the first packet queued would be the first packet to be sent out. There are 11 types of queuing disciplines which are currently supported in Linux.

Each queue has routines which help it initialize itself, enqueue a packet, dequeue a packet, requeue a packet and drop a packet. Each of the queues also have a set of QoS parameters defined which help the queues maintain the traffic based on the QoS parameters. Queues are identified by a handle of the form `<major number:minor number>`. Queuing disciplines can also be classified as classless queuing discipline and classful queuing discipline which will be explained in greater detail once classes have been defined.

Classes: Classes and queuing disciplines are intimately tied together. Each class owns one or more queues. By default on creation, the class owns a FIFO queue, but this can be changed to any other queue type. A class is identified by a class ID which is specified by the user. The kernel maintains an internal identifier for each of the classes which are in use.

Filters: Filters are used to classify packets based on the properties of the packet. For example IP address or port number etc. Filters provide a convinient mechanism for gluing together several of the key elements of the Traffic Control. A filter can be

attached either to a classful queueing discipline or to a class. The filter can redirect the packet into any of the subclasses associated with this filter.

Queueing disciplines can further be distinguished based on their relation with classes. This has been explained in detail below:

Classless Queuing Discipline: A classless queuing discipline is defined as a queuing discipline which can be owned by another class, but it cannot own a class. Hence these queuing disciplines form the leaf nodes of a complex queuing discipline hierarchy. First-In-First-Out (FIFO), Stochastic Fairness Queuing (SFQ), Generalized Random Early Detection (GRED), Token Bucket Filter (TBF) are some of the classless queuing disciplines.

Classful Queuing Discipline: A classful queuing discipline is one which can be owned by a class and can also own a class in turn. Classful queuing disciplines can be used to create complex hierarchical queuing discipline structures to segregate packets and provide the desired Quality of Service. Hierarchal Token Bucket (HTB), Priority (PRIO) and Class Based Queuing (CBQ) are the classful queuing discipline.

After explaining some of the components of the Traffic Control subsystem, we get down to explaining Traffic Control as a whole. Each network device on a Linux system has a queuing discipline associated with it. Any packet needing to be transmitted by this device will be enqueued on its queuing discipline before being transmitted out. Figure 3.3 shows the Traffic Control as a whole system. FIFO is the default queuing discipline which is loaded on start up. This is a classless queuing discipline which enqueues and dequeues packet in a first-in first-out order. Traffic Control provides routines which can be used to replace this default queuing discipline with any other queuing discipline. By loading a classful queuing discipline we can add sub classes to a queuing discipline which will contain queues in turn. Filters are used to distinguish packets based on the characteristics of the packet and to enqueue the packet into different queues. By controlling the way in which these packets are enqueued we can control the way in which the packets are transmitted.

One of the main advantages of the QoS support on Linux is the flexibility with

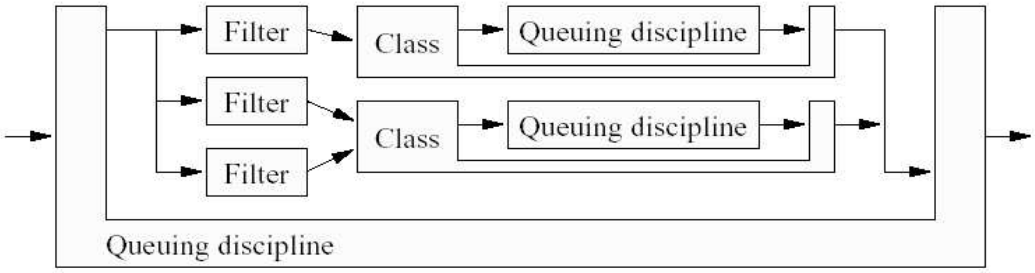

Figure 3.3: Combination of queuing discipline and classes

which the queues and classes can be set up. Each queuing discipline can contain a number of classes. These classes in turn use queues to store packets which can again contain a number of classes. In this way the Traffic Control layer on Linux gives us the flexibility to construct a hierarchy of policy and achieve the desired quality of service.

Traffic Control provides a user level command called ”tc” (Traffic Controller) which can be used to create and associate queues to the output device on a given system. It also lets users to add classes, filters and associates queues to the queuing discipline. The Traffic Control routine also has an option to be compiled as a module in the Linux operating system. Additions and changes to the Traffic Control can be done dynamically after startup without recompiling or restarting the system. This gives us the flexibility to modify the system's QoS as desired without shutting down the system.

3.6 Linux Network Stack

In order to make any changes, we need to first understand the exact implementation of the Linux network protocol stack. Once this is done, we can look into the execution delays that a packet may incur in the present implementation of the network stack and try to reduce the delay with minor modifications. The study of the network stack was done using DSKI instrumentation points. The Linux network stack was instrumented using the DSKI and the flow of control through the kernel was understood. This explanantion also includes names of functions, along with their path relative to the base

installation directory of Linux. The format followed to represent these functions are : `fuction_name` [`file_name`]. We can split up the Linux network stack components into 2 broad categories.

- Transmit side - The code which transmits a locally generated packet out of the system.
- Receive side - The code which receives and processes a packet whose desination is the local system.

Other than this we also have packets which are received, but are not for this system. These packets use parts of the transmit and receive sides. We do not bother with these packets as we are more concerned about packets which go in and out of a local system. We next explain each of the above categories in greater detail.

3.6.1 Transmit packet flow

This section describes packet flow through the kernel starting from when the packet was generated by the Application layer. The packet flow description is segregated based on the layer which processes the packet. We also split the processing into points which can be correlated to Figure 3.4 which gives a pictorial view of the flow of packet on the transmit side.

Application Layer

Step 1: Application writes to the socket using a socket system call. This is done at the application level by the user program. Linux provides many system calls which can be used to 'write to' and 'read from' a socket. Some of the common system calls to send message over a socket are `send`, `sendto`, `sendmsg`, `write` and `writev`. Each of the system call interface routine in the user library, has a corresponding implementation of the function in the kernel.

Step 2: We consider using the `sock_write [net/socket.c]` function here. All the system calls related to writing to a socket finally call the `sock_sendmsg [net/socket.c]`. This function checks if the user buffer space is readable. It gets the socket structure using the file descriptor provided by the user program. It creates a message header structure and fills the data into it. This also creates the socket control message which has few fields which hold the control information like the UID, GID and PID of the process.

Step 3: The control then flows to the INET layer specific function, under the Socket layer which acts like an interface between TCP layer and Socket layer. The INET layer does some validation for the socket structure. It checks for the lower layer protocol pointer and calls the appropriate protocol. This is mainly implemented in the `inet_sendmsg [net/ipv4/af_inet.c]` routine.

Transport Layer (TCP/UDP)

Step 4: The control next flows to the Transport layer. Depending on the protocol used in the Transport layer, TCP or UDP, appropriate functions are called. Here we explain the functions done by both the layers, one after another.

We will talk about the TCP layer first. The `tcp_sendmsg [net/ipv4/tcp.c]` creates the `sk_buff [include/linux/skbuff.h]` structure first. The `sk_buff` structure is the most important structure in the Linux networking stack. Instead of passing the data packet from layer to layer, a reference to this `sk_buff` structure is passed between the layers. In the TCP layer, first the state of the TCP connection is checked. Control waits until the connection is complete, if not completed previously. The previously used `sk_buff` is checked for any tail space available to hold the current data. If found then the same `sk_buff` is used to send the present data, otherwise the data is stored in the new `sk_buff`. It copies the data from the user space to the appropriate `sk_buff` structure. It also computes the checksum of the packet.

In the UDP layer, the `udp_sendmsg [net/ipv4/udp.c]` routine checks the packet length, flags and the protocol used. It then builds the UDP header, at the same time

checking and verifying the fields in the header. It checks if it is a connected socket, if so it sends the packet directly, else it does a route lookup based on the IP address.

Step 5: The `tcp_transmit_skb [net/ipv4/tcp_output.c]` routine builds the TCP header and adds it to the `sk_buff` structure. The checksum is counted and added to the header. It also checks for the ACK and SYN bits. It also checks the header for the IP address, state of the connection and the source, destination port addresses.

The `udp_getfrag [net/ipv4/udp.c]` routine copies the UDP packet from the user space to the kernel space. It then calculates the checksum for that packet. However, this function is also called from the IP layer which initializes the `sk_buff` space for the packet.

Network Layer (IP)

Step 6: The IP layer receives the packet sent from the TCP layer and builds an IP header for it. It also calculates the IP checksum. The `ip_queue_xmit [net/ipv4/ip_output.c]` routine in IP layer does a route lookup, for a TCP packet, based on the destination IP address and figures out the route the packet has to take.

In the case of a UDP connection, the IP layer creates a `sk_buff` structure to store the packet. It then calls the `udp_getfrag` function mentioned above to copy the data from the user space to the kernel space. Once this is done, it directly goes to the Link layer without getting into the next step of fragmentation.

Step 7: The packet is next checked to see if fragmentation of the packet is required, i.e. if the packet size is greater than the permitted size. If fragmentation is needed, then the packets are fragmented in the routine `ip_queue_xmit2 [net/ipv4/ip_output.c]` and sent to the Link layer. This routine is implemented only for a TCP connection.

Data Link Layer

Step 8: The `dev_queue_xmit [net/core/dev.c]` routine in the Data Link layer, receives the packet and completes the checksum calculation if not already done in the

previous layers or if the output device supports a different type of checksum. It checks if the output device has a queue and queues the packet in the output device. It also initiates the scheduler to dequeue the packet and send it out.

Step 9: The `qdisc_run [include/net/pkt_sched.h]` routine checks the device queue for any pending packets which need to be transmitted. If present it initiates the transmission. This function runs in the process context, the first time it is tries to transmit a packet. However, if the device is not free or the process is not able to transmit the packet out for some other reason, then this function is executed again in a softirq context.

Step 10: The `qdisc_restart [net/sched/sch_generic.c]` routine checks to see if the device is free, if so it transmits the packet. If the device is not available or free to transmit, then the transmit softirq, NET_TX_SOFTIRQ is raised.

Step 11: If the device is free, then the `hard_start_xmit [drivers/net/device.c]` is called which transmits the packet out of the system. This routine is a device specific routine and implemented in the device driver code.

Step 12: The packet is sent out to the output medium by calling the I/O instructions to copy the packet to hardware and start transmission. Once the packet is transmitted, it also frees the `sk_buff` space occupied by the packet in the hardware. It also records the time when the transmission took place.

Step 13: This is the path taken if the device is not free to send the packet. In this case the packet is requeued again for processing at a further time. The scheduler calls the `netif_schedule [include/linux/netdevice.h]` function which raises the NET_TX_SOFTIRQ, which would take care of the packet processing at the earliest available time.

Figure 3.4: Network Transmit

Step 14: Once the device finishes sending the packet out it raises a hardirq to inform the system that it has finished sending the packet. If the `sk_buff` is not free at this point of time, then it is freed. It then calls the `netif_wake_queue [include/linux/netdevice.h]` which is basically to inform that the device is free for sending further packets. This function in turn raises a softIrq to schedule the sending of the next packet.

3.6.2 Receive packet flow

Here we explain packet flow through the kernel starting from when the packet was received at the network interface. The receive side of the network stack is more complicated than the transmit side as the control flow is not linear. For example there is control flow from the device layer up the stack and there is control flow from the application layer which initiates the further processing of the packet. We segregate them based on the layer in which the particular processing takes place. We also split the processing into points which can be correlated to Figure 3.5 which gives a pictorial view of the flow of packet on the receive side.

Application Layer

Step 1: The user process reads data from a socket using the read or the variants of receive socket API calls (recv, recvfrom). These functions are mapped onto the `sock_read` and `sys_recvfrom [net/socket.c]` system calls.

Step 2: The system calls setup the message headers and call the `sock_recvmsg [net/socket.c]` function. The `sock_recvmsg` function calls the receive function for the specific socket type, for INET socket type the `inet_recvmsg [net/ipv4/af_inet.c]` routine is called.

Step 3: The `inet_recvmsg` checks if the socket is accepting data and calls the corresponding protocol's receiver function depending on the Transport layer protocol used

by the socket. For TCP it is `tcp_recvmsg [net/ipv4/tcp.c]` and for UDP it is `udp_recvmsg [net/ipv4/udp.c]`.

Transport Layer (TCP/UDP)

Step 4: The TCP receive message routine checks for errors in the socket connection and waits until there is at least one packet available in the socket queue. It cleans up the socket if connection is closed. It calls `memcpy_toiovec [net/core/iovec.c]` to copy payload from the socket buffer to the user space.

Step 5: The UDP receive message routine gets the UDP packet from the queue by calling `skb_recv_datagram [net/core/datagram.c]` routine. It then calls `skb_copy_datagram_iovec [net/core/datagram.c]` to move the payload from the socket buffer to the user space. It also updates the socket timestamp, fills in the source information in the message header and frees the packet memory.

The control flow from the application layer is blocked until data is available to be read by the user process. We proceed ahead by explaining the reception of a packet at the physical layer.

Physical Layer

Step 6: A packet arriving through the medium to the Network Interface Card (NIC) is checked and stored in its RAM. It then transfers the packet to the kernel memory using DMA. The kernel maintains a receive ring-buffer "`rx_ring`" which contains packet descriptors pointing to locations where the received packets can be stored. The NIC then interrupts the CPU to inform about the received packets. The CPU stops its current operation and calls the core interrupt handler to handle the interrupt.

Step 7: This interrupt handling routine, which is device dependent, creates a socket buffer structure (`sk_buff`) to store the received data. The interrupt handler then calls

`netif_rx_schedule [include/linux/netdevice.h]` routine which puts a reference to the device in a queue attached to the interrupted CPU known as the `poll_list`. It also marks for further processing of the packet as a softirq by calling the `cpu_raise_softirq [kernel/softirq.c]` (NET_RX_SOFTIRQ).

Step 8: When the NET_RX_SOFTIRQ softirq is scheduled, it executes its registered handler - the `net_rx_action [net/core/dev.c]` routine. Here the CPU polls the devices present in its `poll_list` to get all the received packets from their `rx_ring` or from the backlog queue, if present. Further interruptions are disabled until all the received packets presents in the `rx_ring` are handled by the softirq. The `process_backlog [net/core/dev.c]` function is assigned as the poll method of each cpu's socket queue's backlog device (`blog_dev`) in the `net_dev_init [net/core/dev.c]` routine. The backlog device is added to the poll list, (if not already present), whenever `netif_rx [net/core/dev.c]` routine is called. This routine is called from within the `net_rx_action [net/core/dev.c]` receive softirq routine, and in turn dequeues packets and passes them for further processing to `netif_receive_skb [net/core/dev.c]` routine.

Step 9: The device's main receive routine is the `netif_receive_skb` which is called from within NET_RX_SOFTIRQ softirq handler. It checks the payload type, and calls any handler(s) registered for that type. For IP traffic, the registered handler is the `ip_rcv [net/ipv4/ip_input.c]` routine.

Network Layer (IP)

Step 10: The main IP receive routine is `ip_rcv` which is called from `netif_receive_skb` when an IP packet is received on an interface. This function examines the packet for errors, removes padding and defragments the packet if necessary. The packet then passes through a pre-routing netfilter hook and then reaches `ip_rcv_finish [net/ipv4/ip_input.c]` routine which obtains the route for the packet.

Step 11: If it is to be locally delivered then the packet is given to `ip_local_deliver [net/ipv4/ip_input.c]` function which in turn calls the `ip_local_deliver_finish [net/ipv4/ip_input.c]` function to send the packet to the appropriate Transport layer function; `tcp_v4_rcv` in case of TCP and `udp_rcv` in case of UDP. If the packet is not for local delivery then the routine to complete packet routing is invoked.

Transport Layer (TCP/UDP)

Step 12: The `tcp_v4_rcv [net/ipv4/tcp_ipv4.c]` function is called from the `ip_local_deliver` function in case the packet received is destined for a TCP process on the same host. This function in turn calls other TCP related functions depending on the state of the connection. If the connection is established it calls the `tcp_rcv_established [net/ipv4/tcp_input.c]` function which checks the connection status and handles the acknowledgements for the received packets. It in turn invokes the `tcp_data_queue [net/ipv4/tcp_input.c]` function which queues the packet in the socket receive queue after validating if the packet is in sequence. This also updates the connection status and wakes the socket by calling the `sock_def_readable [net/core/sock.c]` function. The `tcp_recvmsg` copies the packet from the socket receive queue to the user space.

Step 13: The `udp_rcv [net/ipv4/udp.c]` function is called from the `ip_local_deliver` if the packet is destined to an UDP process in the same machine. This function validates the received UDP packet by checking its header, trimming the packet and verifying the checksum if required. It calls `udp_v4_lookup [net/ipv4/udp.c]` to obtain the destination socket. If no socket is present it sends an ICMP error message and stops, else it invokes the `udp_queue_rcv_skb [net/ipv4/udp.c]` function which updates the UDP status and invokes `sock_queue_rcv_skb [include/net/sock.h]` to put the packet in the socket receive queue. It signals the process that data is available to be read by calling `sock_def_readable [net/core/sock.c]`. The `udp_recvmsg` copies packet from the socket queue to the user space.

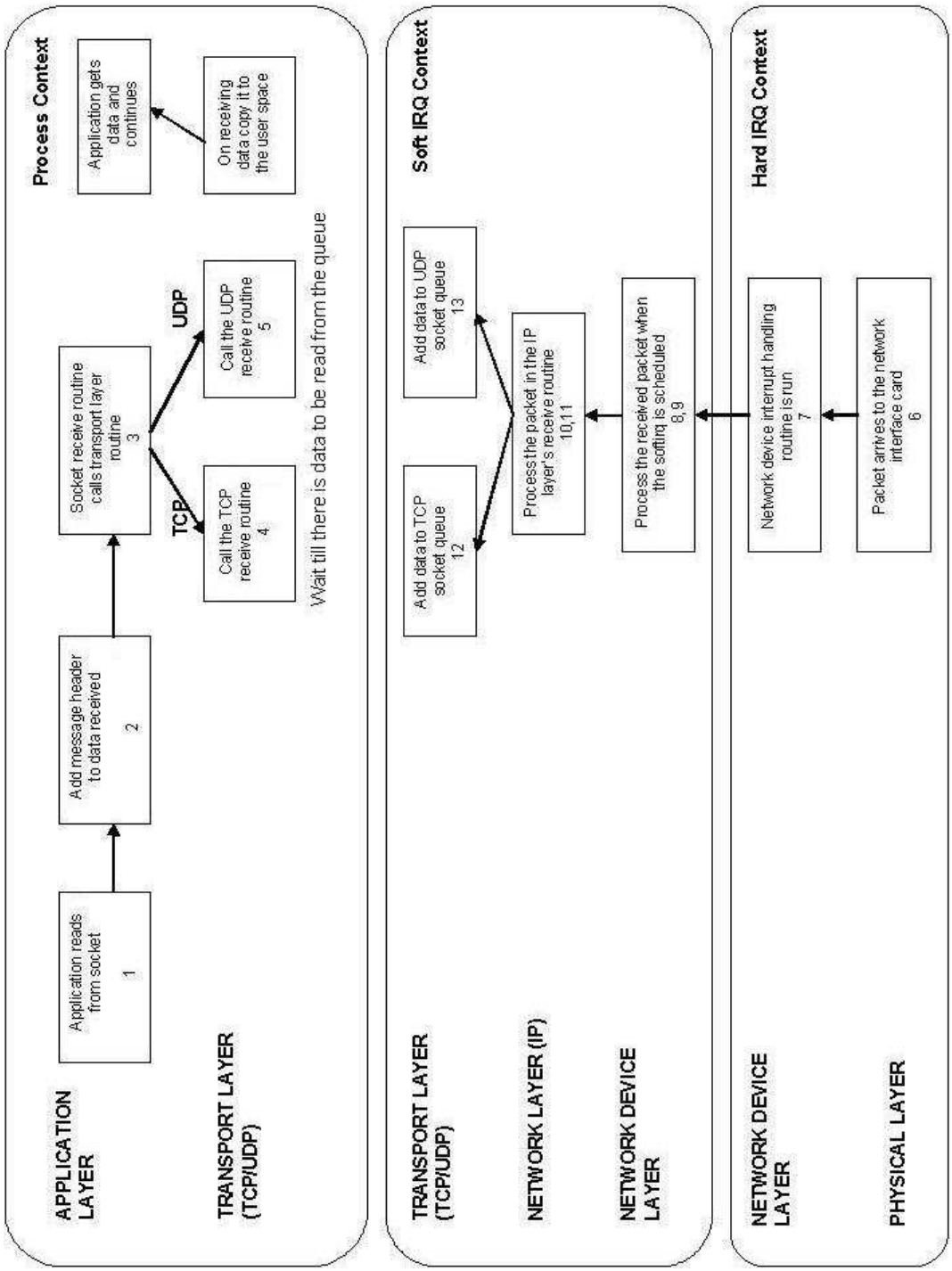

Figure 3.5: Network Receive

Chapter 4

Implementation

The network contributes significant variation to the application behavior because of the lack of determinism in terms of transfer time. In order to make the network suitable for real-time applications, we have to provide QoS such that the end-to-end packet transfer time for a real-time application is determinitstic. We try achieve this through two changes - make the network deterministic in terms of packet transfer time, and reduce the processing time for each packet through the network stack. We achieve a considerable amount of determinism by using Time Division Multiplexing over Ethernet. TDM divides the transmission time into timeslots, which can be used by the systems on the LAN. As each system gets a unique time slice for transmission, only one system transmits at a given time, hence avoiding collisions in the LAN.

In order to make the network more suitable for real-time applications, we have to reduce the packet processing latency, for the real-time applications, in the network stack. There are two main types of packet processing that takes place in the networking stack.

1. Packet processing on the transmit side - The message is split, embedded in a packet and sent out of the system.

2. Packet processing on the receive side - A received packet is processed and defragmented in order to get the message.

The working of the Linux network stack has been explained in detail in section 3.6.

We study the transmit and the receive side of the network stack, to identify the places in the code which could cause delays in packet processing. We also make modifications to the code to include packet queue ordering decisions in the network stack. Using this method, we could select a channel or a particular connection, which needs to receive non-uniform treatment over the rest of the connections.

4.1 Priority in packet processing

Packet processing is the processing that takes place on a packet in the network stack. The packet processing time may be very small compared to network transfer time in most of the scenarios. However, when the system is loaded with lots of applications, or when the system has to send or receive lots of data, the processing time for a packet may be considerable. In order to maintain the performance of the system for a connection in such a scenario, we can give preference to a particular connection or channel. The method of prioritizing the channel includes differentiating the channel from the rest of the channels and processing it before the rest of them. Modifications done in the kernel to prioritize channels and process these channels have been explained in greater detail in Section 4.1.1 and Section 4.1.2

4.1.1 Queue on the Transmit Side

When we study the transmit side, we can see that a packet is processed in the same context from the application layer, through the network stack, until it gets enqueued in the Traffic Control queue. The NET_TX_SOFTIRQ takes over the process at the Traffic Control queue, dequeues the packet and sends it out on the network device. The only delay in processing is the time, when the packet is in the queue, waiting to be dequeued and processed. The dequeuing of the packet takes place in the NET_TX_SOFTIRQ, which is executed only when it is time for transmission i.e. when the time-slot for this system occurs.

We have to choose between two options - one is to disable TDM and transmit the packet immediately, hence reducing the time delay between enqueuing and dequeuing

the packet. This would reduce the time delay in the kernel considerably, but would work on normal Ethernet, thus increasing the transmission delay due to collisions and retransmissions. The other option is to use TDM, which might cause a delay in the kernel, but would provide guaranteed transmission of the packet without collision.We chose to use TDM instead of normal Ethernet. We are compromising with the delay in the transmit processing in order to attain a collision free network.

We could use different methods to uniquely select a connection and prioritize it. We need to select one of the parameters which uniquely identify a connection. We chose the method of using port numbers to uniquely select a connection. In order to prioritize a packet which is being transmitted out of a queue, we need to identify it and process it before the other packets. The simple method to implement this is to use a queue which enqueues all packets for transmission and processes the prioritized packets before the non-prioritized packets. This method is more appropriate in this situation as the packets are already being enqueued in the Traffic Control queue before being processed by the NET_TX_SOFTIRQ. We implemented a new Traffic Control queuing discipline, called the 'TDM queue' to include all the desired features.

TDM Queuing Discipline

Traffic Control was used to implement the queue on the transmit side for many reasons. The first reason being that the packets were already being queued at the Traffic Control queue, so it would be efficient in using the same queue and not having another point of queuing to implement prioritization. Another reason was that the Traffic Control provides a variety of tools which gives the flexibility to add a new queue, change a queue, set Quality of Service (QoS) parameters etc. We could use all these features available in the TDM queuing discipline, to achieve the desired Quality of Service.

The TDM queuing discipline, shown in Figure 4.1, is a simple queuing discipline, which consists of a single First-in First-out queue by default. It has a single queue which enqueues packets on one side and dequeues packets on the other side for transmission. This queuing discipline can be configured to contain more than one queue where the number of queues required is decided by the user. Each queue in the queu-

ing discipline, except for the default queue, corresponds to atleast one port number, i.e. each queue enqueues and dequeues a packet corresponding to a particular port number(s). These queues are First-in First-out queues and are processed one after the other. So the first queue to be processed is the queue with the highest priority and so on. The default queue is always the last queue to be processed. The user can select the port number for each of the queues and the order in which the queues are processed, hence setting up a priority based on the port numbers.

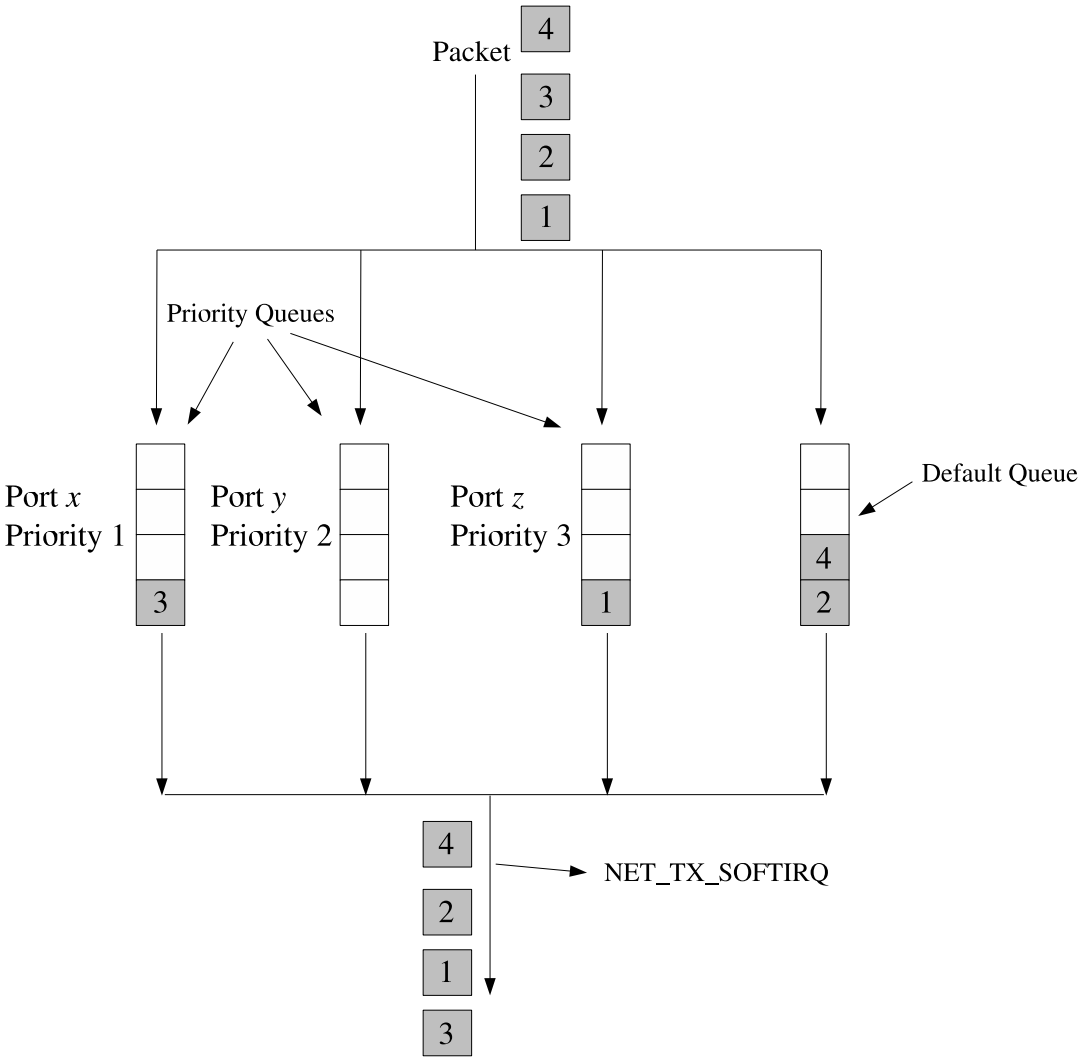

Figure 4.1: TDM Queuing Discipline

4.1.2 Queue on the Receive Side

The receive side of the network stack has been explained in detail in Section 3.6.2. In order to include priority processing on the receive side; we need to devise a method to differentiate a particular connection or channel and prioritize it. Similar to the transmit side, we have used port numbers to uniquely identify a connection on the receive side, and to process the received packets under a configurable policy. There are many ways to implement prioritization of a packet. We have implemented queues on the receive side to differentiate packets and process the prioritized packets.

There were 2 major modification on the receive side to implement prioritization of packets are:

1. Addition of a queuing discipline on the receive side.
2. Breaking up of the receive softirq into 3 separate softirqs.

Section 4.1.3 explains the breaking up of the softirq in detail. Here we discuss the addition of the queue in greater detail.

Adding a queuing discipline on the receive side

A new queuing discipline (set of queues) was added on the receive side to include priority processing, which functions similarly to the one on the transmit side. By default, the queuing discipline has a single First-in First-out queue which enqueues packets on one side and dequeues them on the other side. When the queuing discipline is not configured to provide priority to a particular port number, the packets are not differentiated. All packets are enqueued into the default queue, dequeued, processed and sent up the network stack. This queuing discipline can be configured to contain more than one queue, where each queue is associated with atleast one prioritized port number, along with the default queue for the default packets. Each of the queues corresponds to a particular connection receiving QoS. The packets in the priority queues are processed first, followed by the packets in the default queue. The queue that is processed first, corresponds to the port number which has the highest priority and so on.

The queuing discipline has been designed to work independently of the Time Division Multiplexing feature of the kernel. The queuing discipline has been implemented as a part of the kernel that can be set and reset dynamically at runtime. The user can select the number of queues needed in the queuing discipline, at runtime. The kernel allocates space for these queues, based on the user's selection. Since these queues are implemented independently of the network stack, they can be reset and reloaded with a new set of queues, which are associated with a different set of port numbers.

We have also implemented a user level module which will interact with the queues to set the required parameters from command line. These command line parameters help the user configure and assign port numbers to the queues on the receive side. The details of the command line options have been discussed in greater detail in Section 4.3.

4.1.3 Classification of packets

The NET_RX_SOFTIRQ processes the packet received by the system. This softirq processes the packet, from the hardware device, all the way up the network stack. In order to add a new queue for priority processing on the receive side, this softirq was modified and split into three separate softirqs.

1. NET_RX_SOFTIRQ
2. NET_RX_PRIORITY_PROCESS_SOFTIRQ
3. NET_RX_NORMAL_PROCESS_SOFTIRQ

The NET_RX_SOFTIRQ is split based on its functionality. One of the softirqs is used to classify the packets based on their port numbers, where as the other two are used to process the packets, one to process non real-time packets and the other one to process real-time packets. The functionality of the softirq has been explained in greater detail in Figure 4.2.

NET_RX_SOFTIRQ

The NET_RX_SOFTIRQ is the first softirq to be executed on the receive side after the interrupt service routine. This softirq looks at the packets and classifies them based on their port number. In a Linux system, on reception of a packet, an entry indicating the network device requires service is added to the input queue, without processing the packet. All the processing of the packet is done in softirq context. The NET_RX_SOFTIRQ, polls the device queue on the receive side to dequeue service requirement records one after another and process the packets ready on those devices. The NET_RX_SOFTIRQ has been modified and does not process the packet all the way up to the Transport layer, instead, has the responsibility of classifying the packet and enqueuing it into one of the queues based on the classification. Classification is done based on a data structure maintained in the kernel which contains the list of prioritized port numbers. The NET_RX_SOFTIRQ processes each of the packets from the device queue, checks to see if it is a prioritized packet or not. If it is a prioritized packet, it is enqueued in the appropriate queue; else it is enqueued in the default queue. In this way the NET_RX_SOFTIRQ processing is modified to take the responsibility of classifying the packets and enqueuing it into different queues based on the classification.

NET_RX_PRIORITY_PROCESS_SOFTIRQ

The modified NET_RX_SOFTIRQ does not process a packet all the way up to the Transport layer queue, instead classifies it. NET_RX_SOFTIRQ adds the prioritized packets, which needs to be processed at the earliest, into the appropriate queue, and the rest of the packets into the default queue. NET_RX_PRIORITY_PROCESS_SOFTIRQ was created to complete the packet processing of priority packets after classification. The NET_RX_PRIORITY_PROCESS_SOFTIRQ picks up packets from the prioritized queue and processes it all the way up to the Transport layer. This softirq does not dequeue or process any packet in the default queue. So if there are no priority packets present in the priority queues, or if none of the priority queues are initialized, then this softirq does not process any packets. The NET_RX_PRIORITY_PROCESS_SOFTIRQ

checks for packets in the priority queues, one after another based on the priority, dequeues them, if any are found, and processes them. It then continues to the next queue until it reaches the last queue in the list of priority queues.

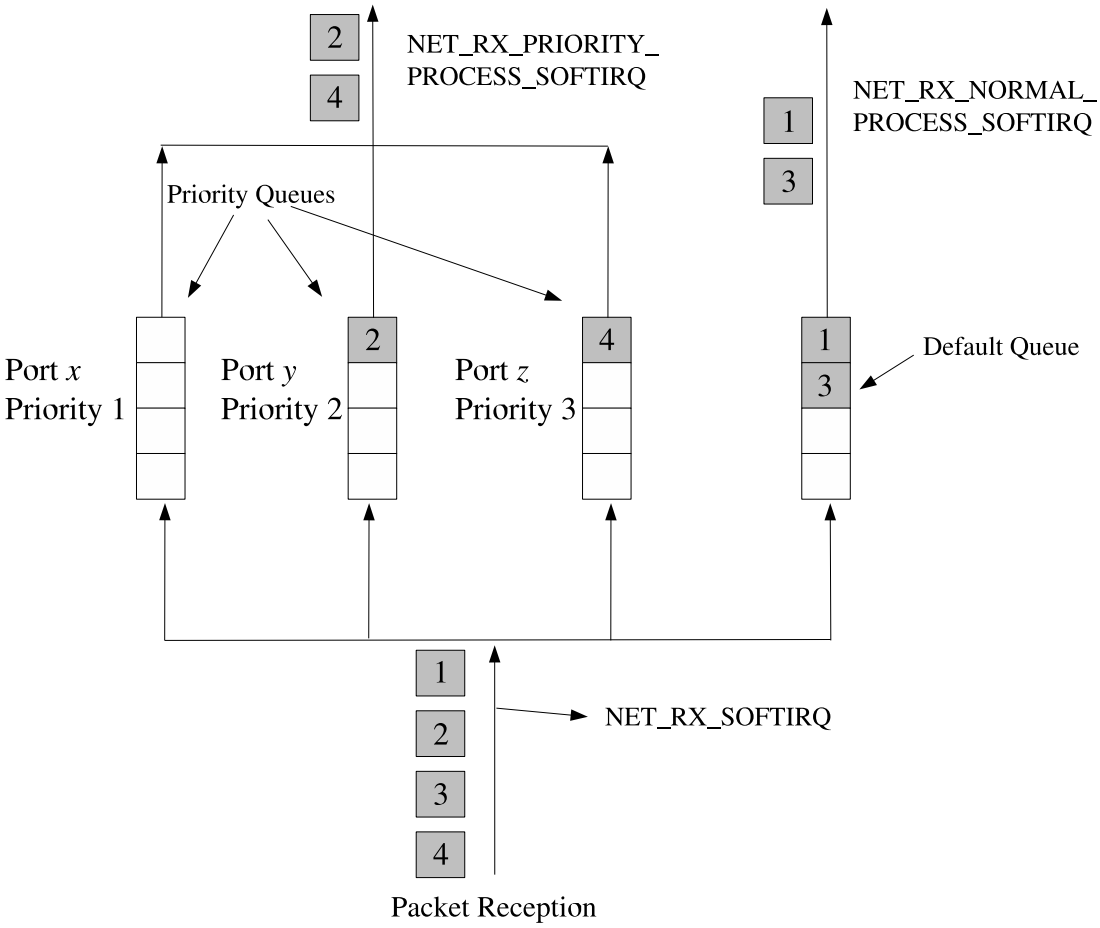

Figure 4.2: Queuing Discipline on the Receive side

NET_RX_NORMAL_PROCESS_SOFTIRQ

This softirq represents the original semantics of the receive softirq. This softirq is similar to the NET_RX_PRIORITY_PROCESS_SOFTIRQ, except that this softirq processes the normal packets rather than the priority packets. As there is only one default queue in our implementation, this softirq checks this default queue for any packets. If the softirq finds a packet in the queue, then it dequeues it and processes it all the

way up to the Transport layer. This softirq does not process any of the priority packets found in the priority queues and only processes the packets in the default queue. The default queue is present in the system on start up. So before the configuration of the priority queues on the system, all packets are classified as normal packets and are added into the default queue. Hence, all the packets are processed by this softirq. Once the priority queues are configured on the receive side, new queues are created for the prioritized port number and these queues are processed by the NET_RX_PRIORITY_PROCESS_SOFTIRQ.

4.2 Group Scheduling Model to achieve Quality of Service

Group Scheduling framework lets us build a hierarchy of groups in order to control the scheduling of processes. It gives us more control over the order in which computational components, which include hardirq, softirq and processes, are selected for execution. This has been explained in greater detail in section 3.4. Here we build upon the TDM Group Scheduling Model to add real-time capabilities to the network stack over Time Division Multiplexed Ethernet.

The Group Scheduling framework has been used in this implementation to reduce the latency in packet processing in the Linux network stack. Section 3.6 gives us a detailed explanation of packet processing in the Linux network stack. We try to reduce this packet processing time, thus giving the network stack, real-time capabilities. We use the Group Scheduling framework to control the sequence of processing of the packets in the kernel to achieve better packet processing latency. The modifications to the Group Scheduling framework fall into two broad categories which is explained here.

Modifications to the Group Scheduling framework to improve performance on the transmit side

When we study the packet processing on the transmit side in the Linux network stack, we can note that the processing is done in the process context until the packet

is enqueued in the Traffic Control queue. The NET_TX_SOFTIRQ dequeues the packet in the softirq context, processes it and transmits it out of the device. This would imply that a packet is processed without any delay on the transmit side till it reaches the Traffic Control queue. The only possible delay on the transmit side is when the packet waits in the Traffic Control queue to be processed by the NET_TX_SOFTIRQ. Once it is enqueued in the traffic control queue, the NET_TX_SOFTIRQ processes it as scheduled. This does not leave much scope for improvement in performance. We could improve the priority of the process, so that it gets executed before the other processes. We could also increase the priority of the NET_TX_SOFTIRQ or schedule the NET_TX_SOFTIRQ before the rest of the softirqs. But the problem with this setup is that, we are currently using TDM on Ethernet as the protocol for the Physical layer, so every system has a fixed time slot when a packet can be sent out. So increasing the priority of the softirq is not going to help us in improving the performance of the packet on the transmit side, unless it is the system's time slot to transmit. By looking at Figure 3.1 we can observe that the TDM Group Scheduling model gives the NET_TX_SOFTIRQ the highest priority during the transmit time slot, which is exactly the desired change. So, no modifications have been done to the Group Scheduling framework used by TDM on the transmit side.

Modifications to the Group Scheduling framework to improve performance on the receive side

When we study the packet processing on the receive side in Section 3.6.2, we can note that the processing sequence has been split into two sections. The processing of each of these sections is initiated from different layers in the network stack. The processing of one of the sections is initiated from the Application layer where the user-level application, which initiates the packet reception, waits on the packet at the Transport layer. This application process runs when the process gets scheduled, which might vary depending on the priority of the process. The processing of the other section is initiated from the Physical layer on reception of a packet. This process is executed in the softirq context and hence proceeds without any delay after being scheduled. There

is scope for delay in packet processing between the time when the packet is added to the Transport layer queue by the softirq and when the application process reads it out of the queue. The application process can get delayed in getting scheduled due to many reasons causing the packet to wait in this queue, hence increasing the processing time. We can reduce this delay by scheduling the application process at an earliest possible time, once the packet reaches the Transport layer queue.

We try to achieve this by using the Group Scheduling framework to schedule softirqs and processes. We add another group called the 'Priority Group' to the Group Scheduling framework that is positioned after the softirqs in the scheduling hierarchy. The Group Scheduling routine would schedule the processes in this group, once it has finished executing all the pending softirqs. Once a packet arrives at a system, the receive softirqs which include all the three softirqs, process the packet all the way up to the Transport layer. The Group Scheduling routine, will execute any of the other pending softirqs, and then schedule the application process in the 'Priority Group', which corresponds to the process waiting for the packet. Using this model we can reduce the waiting time of the packet in the Transport layer queue. The Group Scheduling framework model used by real-time networking model is explained in greater detail later in this section.

The processing done by the receive softirq on the receive side has been split into three separate softirqs. The NET_RX_SOFTIRQ classifies the packet and enqueues the packet in the appropriate receive queue. The NET_RX_PRIORITY_PROCESS_SOFTIRQ processes the prioritized packets up to the Transport layer. The NET_RX_NORMAL_PROCESS_SOFTIRQ processes the normal packets or the packets in the default queue, up to the Transport layer. The softirqs were split into in order to differentiate the packets based on their priority and processes them accordingly. The method of differentiating the packets is done by the NET_RX_SOFITRQ which classifies the packets and enqueues them in different queues. The method of processing the different packets based on their priority can be achieved by changing the priority of the NET_RX_PRIORITY_PROCESS_SOFTIRQ such that the prioritized packets are scheduled at an earliest possible time.

We try to achieve this by changing the priority of the softirqs using the Group Scheduling framework. NET_RX_SOFTIRQ has to have the highest priority among the three receive softirqs, as it needs to be processed first to classify the packets. The NET_RX_PRIORITY_PROCESS_SOFTIRQ has to be scheduled next to process the priority packets, if any present. The NET_RX_NORMAL_PROCESS_SOFTIRQ can be scheduled any time after the priority packets are processed as processing of normal packets can be delayed. The exact priority of these softirqs and the Group Scheduling hierarchy used is explained in detail later in this section.

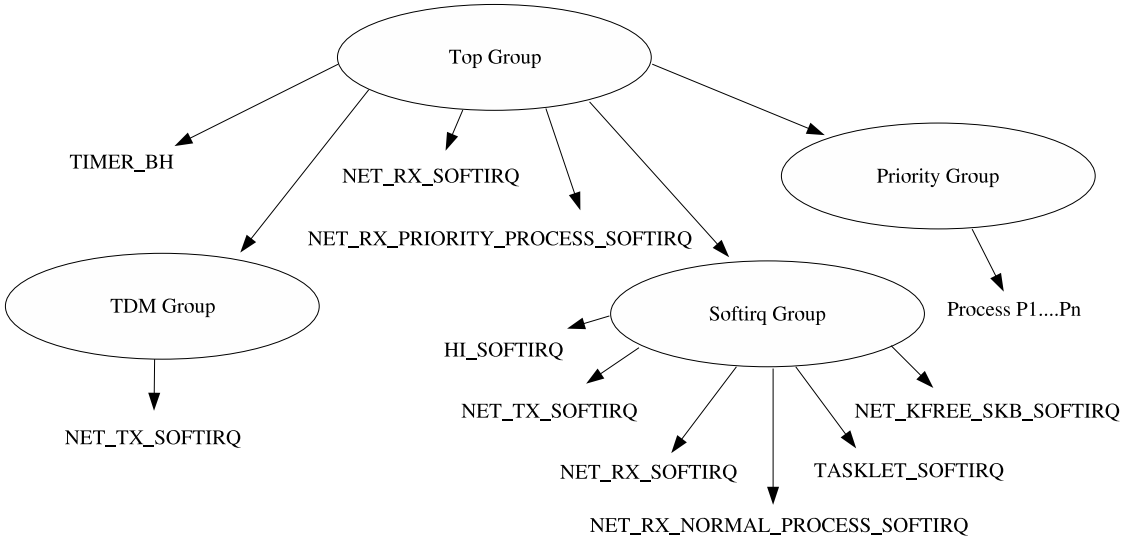

Figure 4.3: Group Scheduling model for Real-Time Networking

The Real-Time Networking model of the Group Scheduling framework was built on the Time Division Multiplexing model. This model is explained in detail in Section 3.4. The Top group contains two sub groups - 'TDM group', which uses the TDM scheduler to control the transmission of packets in the appropriate time slots and 'Softirq group', which processes the remaining softirqs. The modifications made to the TDM model to achieve real-time capabilities falls into two main categories.

Addition of 'Priority Group' - We can notice from the Figure 4.3, that 'Priority Group' is added to schedule prioritized or real-time processes. When a packet arrives on the system, it is processed in softirq context by the receive softirqs and then gets added to the Transport layer queue. The scheduling hierarchy processes the remaining softirqs in the order of their priority and then starts scheduling processes in the 'Priority Group'. This would assure that the process, for which the packet is waiting in the Transport layer queue, gets scheduled at an earliest possible time. This would reduce the waiting time of the packet in the Transport layer queue, thus reducing the delay in packet processing.

The Priority Group can be placed anywhere after the NET_RX_PRIORITY_PROCESS_GROUP, in the Real-Time Networking Group Scheduling model. A design decision needs to be done regarding the placement of the 'Priority Group'. Placing it immediately after the priority process softirq, schedules the processes in the 'Priority Group' even before scheduling the remaining softirqs. Other options include placing the 'Priority Group' in the middle of the softirq group so that it can get scheduled between the QoS network and non-priority network. Looking at Figure 4.3 we can see that we have different options for placing the 'Priority Group'. We have chosen to place the 'Priority Group' after all the softirqs, as we wanted to process the softirqs at the earliest possible time. We assume that the processes in the Priority Group can accommodate this amount of delay in processing.

Placement of Receive Softirq - We can also notice from Figure 4.3, that the position and hence the priority of the receive softirqs has been changed. We can note that the priority of the NET_RX_SOFTIRQ has been increased, as this softirq needs to be processed first for a packet to be received and classified. After classification, a packet can be identified as a prioritized or a normal packet. We can notice that the next softirq which is processed after the receive softirq is the NET_RX_PRIORITY_PROCESS_SOFTIRQ which processes the prioritized packets, if any are present in the queue, at an earliest possible time. We also notice that the priority of the NET_RX_NORMAL_PROCESS_SOFTIRQ is unaltered, as the normal packets can be processed

at a later point of time.

4.3 User Interface

Once the modifications to the kernel were complete, we have to provide a user-friendly method to access and configure the kernel with the required Real-time parameters. The method used by the TDM module to configure the kernel was used, as the Real-Time Networking system was an enhancement to this module. TDM uses a loadable module, which needs to be inserted into the kernel on startup. The user can use a user-level program, which takes command line options, to set, start and stop TDM in a LAN. Routines have been added to this module and the user-level program to enhance it, so that it could be used to configure the Real-Time Networking parameters in the kernel.There are three enhancements added to the TDM module and the user-level program that have been explained in the following sections.

4.3.1 Setting Priority on Transmit side

Traffic Control is being used to implement priority on the transmit side. A Traffic Control queuing discipline called the TDM queue is used for this purpose. This queuing discipline in turn is made up of a number of queues. Each of these queues is associated with atleast one port number, where a port number uniquely identifies a connection. A command line utility is provided, which lets the user, set the port numbers associated with these queues. The command line parameters are:

```
tdm send <# of queues> <space-delimited list (port# priority)>
```

'tdm' is the user-level program which is used. 'send' is the parameter, which specifies to the user-level program that the user is trying to associate port numbers with the transmit queues, followed by the number of queues that need to be created. This is followed by a list of space-delimited port number, which need to be prioritized, and the priority associated with this port number. Each of these port numbers is associated with a queue with the priority as specified by the user.

```
bash> tdm send 3 15001 2 15002 4 15003 1
```

This example creates three queues and sets three prioritized ports on the transmit side of the system - 15001, 15002, and 15003. Port number 15001 is assigned a priority of 2, 15002 is assigned a priority of 4 and the port number 15003 is assigned a priority of 1. We need to note that, a lower number corresponds to a higher priority. So in this example, port number 15003 has the highest priority followed by 15001 and 15002.

4.3.2 Setting Priority on Receive side

A queuing discipline is being used on the receive side to implement priority. This queuing discipline in turn is made up of a number of queues. These queues are used to classify packets on the receive side and process the prioritized packet at the earliest. Each of these queues is associated with atleast a port number, where the port number reflects a connection. A command line utility is provided, which lets the user set the port numbers associated with the queues on the receive side. The command line parameters are:

```
tdm recv  <# of queues> <space-delimited list (port# priority)>
```

'tdm' is the user-level program which is used. 'recv' is the parameter, which specifies to the user-level program that the user is trying to associate port numbers with the receive queues, followed by the number of queues that need to be created. This is followed by a list of space-delimited port number, which need to be prioritized, and priority associated with this port number. Each of these port numbers is associated with a queue with the priority as specified by the user.

```
bash> tdm recv 3 15004 3 15005 5 15006 2
```

This example creates three queues and sets three prioritized ports on the receive side of the system - 15004, 15005, and 15006. Port number 15004 is assigned a priority of 3, 15005 is assigned a priority of 5 and port number 15006 is assigned a priority of 2.

We need to note here that the queues on the transmit side and the receive side are independent of each other. The port numbers on the transmit side and on the receive

side could match, but they need not match. On the transmit side, we would set priority to the port numbers corresponding to the port number of prioritized outgoing packets. On the receive side, we would set priority to the port numbers which corresponds to the port number of the prioritized incoming packets.

4.3.3 Add/Remove Real-time process

The 'Priority Group' was created under the Group Scheduling framework to schedule real-time processes, to reduce the delay in packet processing on the receive side. Adding a process to this group would ensure that the process would get scheduled once all the pending softirqs are processed. This would reduce the waiting time of a packet in the Transport layer queue; hence reduce the packet processing time. A command line utility is provided that let the user add or remove a process from this group. The command line parameters are

```
tdm add process <space-delimited list of process id>
tdm remove process <space-delimited list of member id>
```

'tdm' is the user-level program which is used. 'add process' is the parameter, which specifies to the user-level program that the user is trying to add a process to the 'Priority Group' and 'remove process' is the parameter which specifies to the user-level program that the user is trying to remove a process from the 'Priority Group'. The user also needs to provide a list of processes, which needs to be added while using the 'add process' option. The processes, which need to be added to the 'Priority Group', are input as a list of space delimited process IDs or PID. These processes are added to the 'Priority Group' sequentially. The user also needs to provide a list of processes, which need to be removed while using the 'remove process' option. The processes, which need to be removed from the 'Priority Group', are input as a list of space delimited member IDs. The member ID is the unique ID which is used to identify a member of the Group Scheduling hierarchy. The member ID for a process is returned to the user when he adds a process to the 'Priority Group'. These processes, if present, are removed from the 'Priority Group'.

```
bash> tdm add process 1219 1251
bash> tdm remove process 12 56
```

The example above adds the processes with PIDs 1219 and 1251 to the 'Priority Group'. The second example removes processes with member IDs 12 and 56 from the 'Priority Group'.

Chapter 5

Evaluation

Once the modifications to the kernel and the Group Scheduling model were completed, we needed to test the system for correctness. We need to verify the working of the system and measure its performance as compared to the original system. In this chapter we discuss the testing techniques used to test the performance of the system and evaluate it.

5.1 End-to-End Quality of Service

This test measures the end-to-end packet transfer time between two processes which are running on two different systems that are on TDM based Ethernet. As previously mentioned, TDM does not differentiate between real-time and non real-time processes and only considers the end-to-end packet transfer time of packets awaiting transmission. The modifications to the kernel and the Group Scheduling framework presented in this thesis were done to achieve the required Quality of Service for real-time applications.

The end-to-end packet transfer time is a reasonable metric, which is used to measure the performance of the QoS provided. This could also be expressed in terms of packet processing times on the transmit and receive systems. When we consider a single packet which is being transferred between two applications on different systems, we can consider the time taken for packet processing on the transmitting system,

time taken for the packet to propagate to the receiving system and the time taken for the receiving system to process the packet. Processing on the transmitting system includes the packet processing in the kernel. The packet propagation time includes the time taken for transmitting the packet to the network and for the packet to propagate through the network to the receiving system. Processing on the receiving system is the processing done in the kernel, which includes the packet processing in the softirq and the processing contexts.

There are many factors in the LAN which affect the performance of this test. Factors such as LAN speed (10/100 Mbps), hub or switch, number of systems in the LAN and accuracy of clock synchronization affect this configuration. The TDM schedule decides when a system in the LAN gets to transmit a packet, and hence affects the end-to-end packet transfer time. The other factors which affect the performance of the test includes the number and types of processes running on each of the systems involved in transfer of the packet. In this test scenario, we evaluate the performance of a single real-time process by measuring its end-to-end packet transfer time. We also evaluate the performance of a real-time process with other non real-time processes transfering packets.

The testing involved 4 systems, which were in a LAN setup using a 100Mbps hub. All the systems were using TDM as the MAC layer protocol to communicate with each other. In this test, the TDM schedule used had a total transmission cycle of 1040µs with 4 individual time slots of 260µs (220µs of transmission time and 40µs of buffer time between transmissions) each. All the messages transmitted by both the processes were 64 bytes long. One of the four systems acted as the Time synchronization master and kept the time on all the systems synchronized. Two of the three systems were involved in packet transmission, whereas the fourth system was used to generate other non real-time traffic in the network. The testing was done using a client-server application. The server starts and waits for the client to connect to the server and initiate a packet transfer. Once the client starts execution, it connects to the server and transmits a packet to the server. The server receives the packet, processes it and transmits it back to the client. Once the client receives the processed packet, it sends another packet and

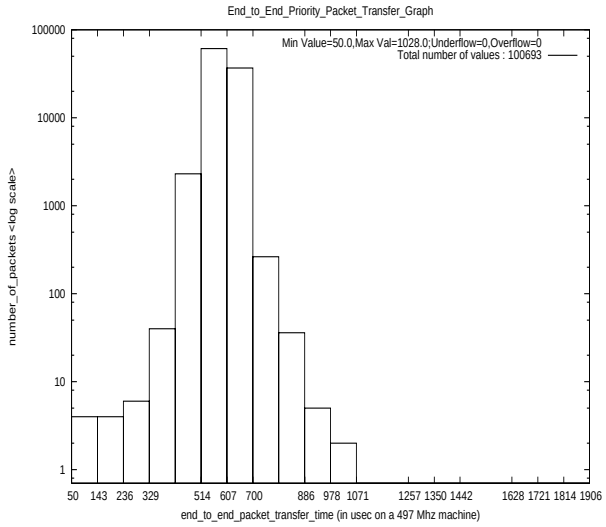

Figure 5.1: End-to-End packet transfer time for a single real-time process

repeats the process for more packets. Time-stamps were collected at different points in the kernel and user space, which marks the flow of packet through the network stack. The time-stamps were used to calculate the packet processing time and the end-to-end packet transfer time.

The Data Stream Kernel Interface (DSKI) and Data Stream User Interface (DSUI) were used to log events at different points in the kernel and user space along the end-to-end execution path. The log events also include a time-stamp, which gives the time of occurrence of the event and can be used for calculating the performance of the system. DSUI events were logged in the user space in the application program. DSUI events EVENT_SERVER_SEND_DATA and EVENT_SERVER_RECV_DATA were logged every time the server application sent and received a packet from the client system respectively. Similarly, events EVENT_CLIENT_SEND_DATA and EVENT_CLIENT_RECV_DATA were logged every time the client application sent and received a packet from the server application respectively. DSKI events were logged in the kernel to record the time of transmission and reception of a packet at a system. The log event HARD_START_XMIT was logged when a packet was transmitted out of a system. On the receiving side, EVENT_CLASSIFY_PACKET was logged which signifies the reception of a packet. The tag field of these events gives the port number of the

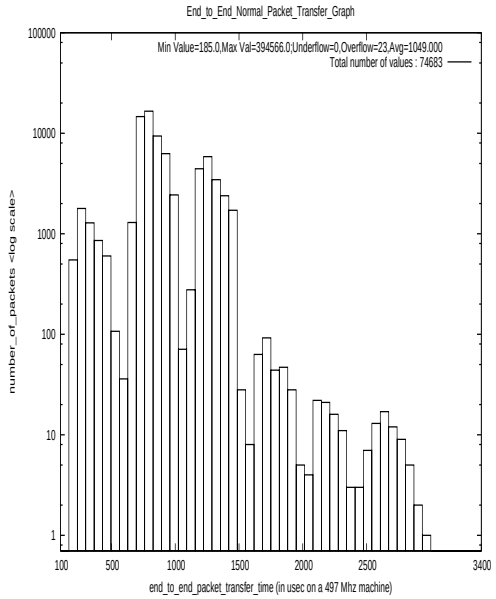

Figure 5.2: End-to-End packet transfer time for a non real-time process

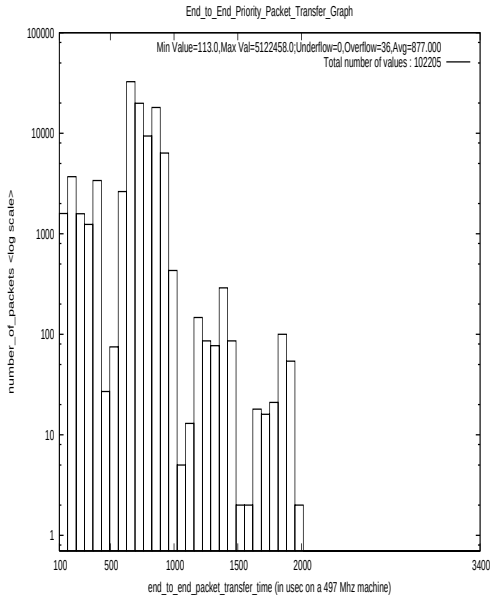

Figure 5.3: End-to-End packet transfer time for a real-time process

packet which caused these events to be logged. We use DSKI kernel filters to only log events which were relevant to the port number we were using, as this number identifies a connection between the systems.

These events were logged and analyzed to know the different processing time for a packet. Post processing filters applied to the DSKI data from different machines remapped these events onto a common global timeline [23]. Other DSKI filters were used to calculate the various values needed to evaluate this system. The difference between the EVENT_CLIENT_SEND_DATA and the EVENT_SERVER_RECV_DATA, gives the end-to-end packet transfer time for a packet. These end-to-end packet transfer times were used to compare the performance of the system for real-time and non real-time processes.

For the first test, we executed a single real-time process in this environment. This bascially illustrates a real-time process scenario, where the packet transfer rate is not high, but packet delay and packet loss are not tolerated. The results of this test have been shown in Figure 5.1. From the histogram we can see that all the real-time packets

transmitted have an end-to-end transfer time of less than 1040µs which corresponds to the total transmission cycle used in this experiment.

Table 5.1: End-to-End packet transfer time

Message Size (Bytes)	Total Slot time (µs)	Average Time (µs)
64	1040	592
256	1200	698
1472	1920	1244

For the second test, we executed a single real-time process along with other non real-time processes and measured its performance. We also ran a non real-time process in the same scenario and measured its performance. From the output shown in Figure 5.3, we can note that most of the real-time packets have an end-to-end transfer time of less than 1040µs. We can also compare this with the end-to-end transfer time for packets from a non real-time application shown in Figure 5.2. We can note that the end-to-end packet transfer time for a non real-time process is almost equally distributed between the first and the second time slot in the TDM schedule.

The packet processing time on the transmit and receive system was calculated for a real-time process. This is displayed in Figure 5.4, Figure 5.5 and Figure 5.6. We can note that the amount of variation in packet processing time on the receive side, as displayed in Figure 5.5, is minimal compared to the packet processing time on transmit side, as displayed in Figure 5.4. We can also note that the variation in packet processing time on the transmit side before enqueuing it in the Traffic Control queue, displayed in Figure 5.6 is also minimal. This implies that the the variation on the transmit side is mainly due to the wait for the transmission time slot.

The other test that was conducted was the end-to-end packet transfer time for different TDM slot times. Tests were conducted with three different slot sizes based on the size of the packet used for transmission [19]. The results of the test have been summarized in Table 5.1. For these tests we could observe that the end-to-end packet transfer time varied based on to the total transmission cycle used for the TDM schedule. The average end-to-end packet transfer time for a packet was close to half of the

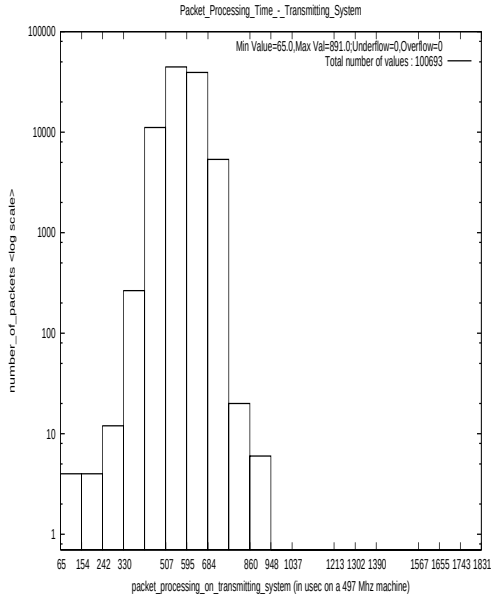

Figure 5.4: Packet processing time on transmit system

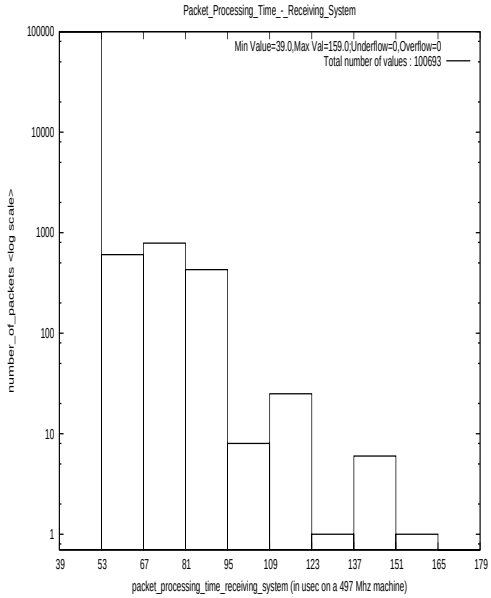

Figure 5.5: Packet processing time on receive system

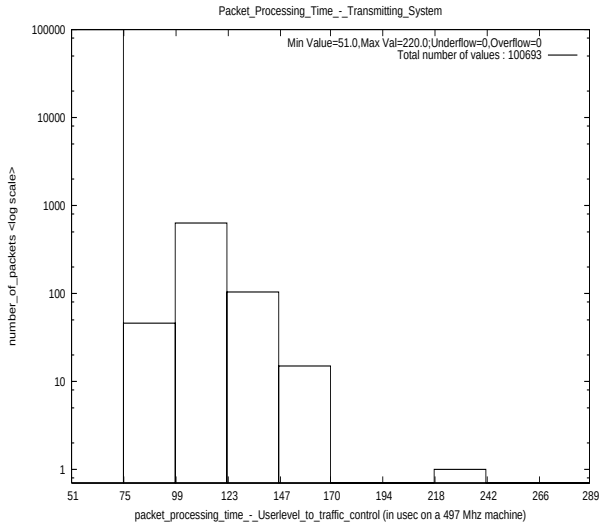

Figure 5.6: Packet processing time on transmit side - user process to Traffic Control

total transmission cycle used for transmission.

5.2 Pipeline Computation

This test was used to calculate the performance of the system for an n-pipe computation where we have 'n' machines lined up for computation and each machine does a part of the computation. This is more of a distributed computation scenario, where the computation is distributed among different systems in a LAN.

There are a few factors that affect the performance of the system for this test scenario. Factors such as LAN speed (10/100 Mbps), hub or switch, number of systems and the accuracy of clock synchronization affect this configuration. The other factors, which affect this test, include the number of computational components that are spread across computers.

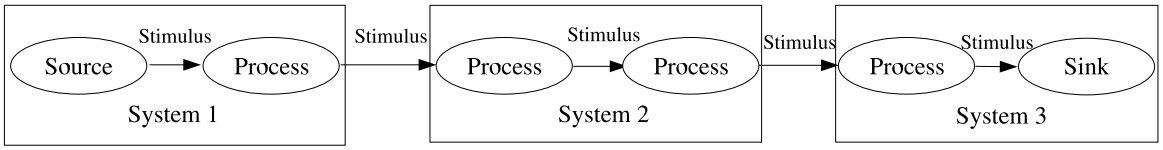

Figure 5.7: Pipeline Computation

The testing involved four systems, which were in a LAN setup using a 100Mbps hub. All the systems were using TDM as the MAC layer protocol to communicate with each other. The individual time slots were 260µs (220µs of transmission time and 40µs of buffer time between transmissions), with a total transmission cycle of 1040µs. All the messages transmitted by all the computations in the pipeline were 10 bytes long. One of the four systems acted as the Time synchronization master and kept the local clocks on all the systems synchronized. The other three systems formed three nodes of the pipeline computation. The pipeline computation was used to calculate the packet processing time in the kernel. This experiment was also visualized using the Datastream visualizer to get a better idea of the sequence of events in the pipeline processing.

The pipeline computation is illustrated in Figure 5.7. Each node contained two

pipeline computation components. The computation starts in the source node, which generates a stimulus that it uses for computation. Once the computations on a particular system completes, it moves to the next system by passing on the stimulus to the next system. This process continues until it reaches the sink system, which ends the pipeline computation. This emulates a distributed computation scenario, where a computation is split into smaller parts that are computed on different systems to distribute the load. Computations on all the systems need to finish to get the end result.

The Data Stream Kernel Interface (DSKI) and the Data Stream User Interface (DSUI) were used to log events at different points in the kernel and user space along the end-to-end computation path. The log events also include a time-stamp, which gives the time of occurrence of the event and can be used for calculating the performance of the system. DSUI events were logged in the user space in the application program. DSUI log events EVENT_START_CYCLE and EVENT_END_CYCLE were used to log events at the start and end of each computation. EVENT_STIMULUS_SENT and EVENT_STIMULUS_RECD were used to log events at the time when a stimulus was sent and received from a particular system to another computation on the same system. Events REMOTE_STIMULUS_SENT and REMOTE_STIMULUS_RECD were used to log events at the time when a stimulus was sent and received between computations that were on different systems.

Other than the DSUI events, DSKI events were logged at different points in the kernel. HARD_START_XMIT was logged in the device driver when a packet was transmitted out of the system. On the receiving side, EVENT_CLASSIFY_PACKET was logged which signifies the reception of the packet. The tag field of these events gives the port number of the packet which caused these events to be logged. We use DSKI kernel filters to log events which were relevant to the port number we were using. In this experiment, the port number identifies a connection between two systems.

These events were logged and analyzed to know the different processing time for a packet. Post processing filters applied to the DSKI data from different machines remapped these events onto a common global timeline [23]. Other DSKI filters were used to calculate the values of the various performance metrics needed for this test.

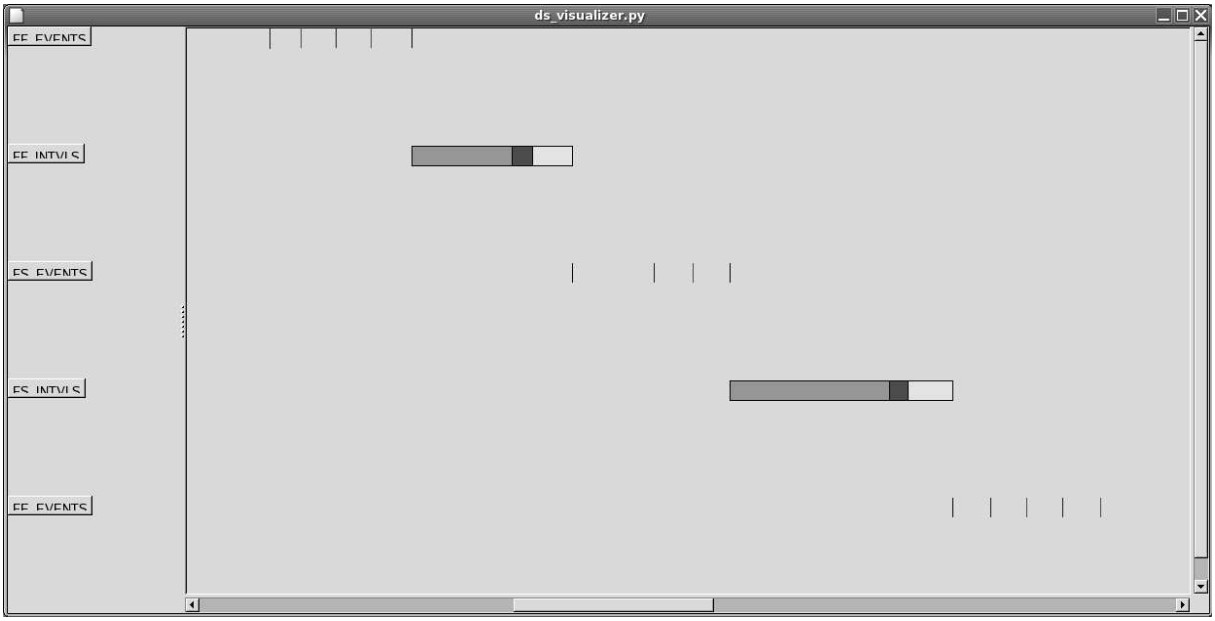

Figure 5.8: Pipeline Computation Visualization

The time difference between REMOTE_STIMULUS_SENT and HARD_START_XMIT, gives the packet processing time on the transmit side. The time difference between REMOTE_STIMULUS_RECD and EVENT_CLASSIFY_PACKET, gives the time taken for packet processing on the receive side. Similarly, the time difference between the events REMOTE_STIMULUS_SEND and REMOTE_STIMULUS_RECD gives the end-to-end packet transfer time for the stimulus. The average packet processing time for the packets on the transmit and receive side have been shown in Table 5.2.

Table 5.2: Packet processing time

	Average Time (μs)
Packet processing time on Transmit side	492
Packet processing time on receive side	41
End-to-End transfer time	537

The postprocessed output was fed into a Datastream Visualizer to visualize the events on a global timeline. The output of the visualizer is shown in Figure 5.8. The visualizer was configured to show the execution timelines for all pipelines. The different lines in the output represent the different events that occurred during different point

of time. The packet processing time on the transmit and receive side are represented as intervals on the visualizer.

Chapter 6

Conclusions and Future Work

Time Division Multiplexing over Ethernet provides a good collision free protocol for a LAN. This protocol deals only with the MAC layer and does not provide any features to differentiate between the processes running on each of the system. This thesis work intended to provide an enhancement in this TDM model so that it can differentiate between the real-time and non real-time processes running on each of the system and provide the appropriate Quality of Service (QoS) as required by the applications. The test results have verified that the enhancements added to the TDM model do provide the required QoS to the real-time processes.

Time constraint in Quality of Service

The method used to provide priority in this project, works independently of the TDM model and the process seeking priority. The method used does not associate a real-time process with any values. It only differentiates between a real-time and a non real-time process, and tries to provide resources to the real-time process to perform better. This implementation does not interact or negotiate with the process using any Quality of Service parameters.

We could extent this project to include a set of QoS parameters, which could be exchanged between the process and the system before the system starts providing QoS. Each process should be aware of the deadline it is trying to achieve and negotiate with the system based on this deadline. The system should maintain the list of real-time

processes running on this system and the QoS parameters of each of these processes. The system should be able to judge whether it would be able to provide the desired QoS, based on the number and QoS parameters of each of the real-time process running on the system. The system should not commit to provide QoS to a process when it cannot achieve it.

The Quality of Service parameters could also be applied to a single message, where each message transmitted or received at a system, has a deadline associated with it, and the system should strive to achieve this deadline.

Resource Reservation Protocol

The Resource Reservation Protocol (RSVP) [3] is used to enhance the current Internet architecture with support for Quality of Service. The RSVP protocol is used by a host to request for a specific QoS from the network, on behalf of an application data stream. This request is done along the reverse data path. This protocol carries this request through the network, visiting each node on the path requesting the desired QoS. This protocol tries to reserve resource on each node for this particular connection so that the connection has the requested Quality of Service.

The Quality of Service discussed in this thesis work, is confined to the particular LAN where it is implemented. This allocates resources in each of the host systems involved in the transfer as in a LAN, message transfer takes place directly between two nodes without a third intermediate node. We can enhance this project so that the QoS module on each of the systems interacts with the RSVP protocol to reserve resources on nodes beyond the LAN. In this way we can extend the Quality of Service provided by this project to go beyond a single LAN.

Bibliography

[1] Linux cross reference. http://lxr.linux.no/source.

[2] Werner Almesberger. Linux network traffic control - implementation overview. Technical report, April 1999.

[3] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (rsvp). RFC 2205, Septmeber 1997.

[4] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The Datastream Kernel Interface (Revision A). Technical Report ITTC-FY98-TR-11510-04, Information and Telecommunication Technology Center, University of Kansas, 1994 June.

[5] Tzi-cker Chiueh Chitra Venkatramani. Supporting real-time traffic on ethernet. In *Proceedings of IEEE Real-Time Traffic Systems Symposium*, December 1994.

[6] Curtiss-Wright Controls Embedded Computing. Scramnet+ shared memory speed, determinism, reliability, and flexibility for distributed real-time systems. http://www.systran.com/ftp/literature/sc/scsmwp.pdf.

[7] Stephen D Cote. Token-ring architecture. http://www.bralyn.net/techpages/papers/token.ring.html.

[8] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel*. 2002.

[9] Will Dinkel, Douglas Niehaus, Michael Frisbie, and Jacob Woltersdorf. *KURT-Linux User Manual*, 2002.

[10] L. Torvalds et al. The Linux Kernel Archives. http://www.kernel.org.

[11] Michael Frisbie. A unified scheduling model for precise computation control. Master's thesis, University of Kansas, March 2004.

[12] Leon Garcia and Widjaja. *Communication Networks*. McGraw Hill, 2000.

[13] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus. Temporal Resolution and Real-Time Extensions to Linux. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, University of Kansas, June 1998.

[14] R. Jonkman, D. Niehaus, J. Evans, and V. Frost. NetSpec: A Network Performance Evaluation Tool. Technical Report ITTC-FY98-TR-10980-28, Information and Telecommunication Technology Center, University of Kansas, December 1998.

[15] Jan Kiska. Rtnet - hard real-time protocol for rtai/linux. www.rts.uni-hannover.de/rtnet/index.html.

[16] Radhakrishnan R. Mukkai. Design of the new and improved netspec controller. Master's thesis, University of Kansas, December 2003.

[17] Paolo Gai Paulo Pedreiras, Lus Almeida. The ftt-ethernet protocol: Merging flexibility, timeliness and efficiency. In *14th Euromicro Conference on Real-Time Systems*, 2002.

[18] Miguel Rio, Mathieu Goutelle, Tom Kelly, Richard Hughes-Jones, Jean Philippe Martin-Flatin, and Yee-Ting Li. A map of the networking code in linux kernel 2.4.20, technical report datatag-2004-1. Technical report, March 2004.

[19] Hariprasad Sampathkumar. Using time division multiplexing to support real time networking on ethernet. Master's thesis, University of Kansas, January 2005.

[20] Kang G. Shin Seok-Kyu Kweon, Min-gyu Cho. Soft real-time communication over ethernet with adaptive traffic smoothing.

[21] Chia Shen and Ichiro Mizunuma. Real-Time Channel-based Reflective Memory. *IEEE Transactions on Computers*, 49(11):1202–1214, 2000.

[22] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software. In *Real-Time Technology and Applications Symposium*, June 1998.

[23] Hariharan Subramanian. Systems performance evaluation methods for distributed systems using data streams. Master's thesis, University of Kansas, 2004.

[24] Andrew S. Tanenbaum. *Computer Networks 3rd Edition*. Prentice Hall, 1996.

[25] Matthew Wilcox. I'll do it later : Softirqs, tasklets, bottom halves, task queues, work queues and timers.