Carnegie Mellon University
Department of Electrical and Computer Engineering

Network Bandwidth Reservation for Real-Time Service Guarantees

A Thesis Submitted to the
Department of Electrical and Computer Engineering,
Carnegie Mellon University
in Partial Fulfillment of the Requirements
for the degree

Master of Science
in
Electrical and Computer Engineering

by
Sourav Ghosh

May 11 1999

Contents

Abstract

ffhe problem of ffff resource management for real-time and multimedia applications mainly deals with the scheduling of multiple tasks with different timing constraints that are accessing multiple resources. ff resource kernel ff13ff provides timely and protected access to the system resources to the different tasks. ffn this project, ff extend the resource reservation model of the resource kernel to guarantee timely access to network bandwidth. ff discuss network reservation for diverse types of applications with different timeliness and bandwidth requirements. ffhe applications explicitly specify their requirements in terms of volume of data and their timing constraints.

ffur proposed scheme has been implemented on both ffreeffffff and ffff-ffach. ffe find that regulating the transmission of bits on the sending side is relatively straightforward. ffn the receiving side, however, the overhead of detecting and dropping unreserved packets can be high. ffur design ensures early dropping of ineligible packets to minimize disruption with savings of upto

Chapter 1

Overview

This chapter provides an overview of the Resource Kernel approach and its application in the domain of network bandwidth management.

1.1 Introduction

Applications such as aircraft control systems, autonomous vehicles, (manufacturing) process control, live monitoring systems and many interactive multimedia systems require real-time computation, storage and communication services from the operating system in order to be effective. The communication services involve transfer of data across the network with real-time Quality of Service (QoS) guarantees in terms of throughput, delay jitter and loss rate. Many real-time applications are distributed in nature and also need to satisfy end-to-end timing requirements spanning network boundaries. These requirements cannot be supported by the “best-effort” services currently offered in data networks.

A wealth of resource management and network bandwidth schemes exists from which one can draw. Our resource management work builds on and significantly extends established real-time scheduling theory and derived processor reservation work reported in ff1Zff. The work in ff1Zff did not deal with the management of multiple resource types, concurrent accesses to different resources, explicit timeliness control, feedback about resource usage, behavioral control on resource overruns, management of interactions among resource users, and considerations of portability, compatibility and automation. In brief, our resource management scheme supports the abstractions behind real-time priority-based scheduling for periodic activities, and service schemes for aperiodics in that framework [20].

1.2 Goals

The work that we report here has three specific goals]

1. Guarantee both processor cycles and network bandwidth for applications at end-points.
2. Maintain explicit control over the fffff cycles consumed by incoming network packets (which could potentially misbehave).
3. Make routers and switches more scalable by avoiding per-session state management while maintaining timeliness guarantees.

Each of the above three goals will be discussed in greater detail in the following sections.

Some of our goals (such as resource centricity and portability) are similar to those of Microsoft Research's Rialto kernel among others. The reservation model also has its counterparts in network reservation protocols as used in fffff and fffffff. However, the operating system problem seems more complex in one sense since inherently different resource types must be dealt with, while networks essentially deal with one type (namely network packets). In another sense, network reservations must be homogeneous, scalable and efficient, making its realization harder in a different sense.

Despite its origins in real-time scheduling theory, we expect our resource management model to be compatible with resource management schemes with their origins in networks such as fffffff, ffff ^{2}ffff3ff, virtual clocks ff2[ff and lottery schedulingff2“ff. The notion of fairness has long been deemed to be anti-thetic to real-time systems and the management of timeliness ff2

Finally, Blazewicz et al. [Y] have shown that the problem of scheduling activities which need multiple resources simultaneously and have timeliness constraints is NP-complete. In our work, we therefore strive for practical and acceptable alternatives which can guarantee access to different resource types.

1.3 Resource Kernels

A *resource kernel* [20] is defined to be one which provides timely, guaranteed and protected access to system resources. The resource kernel allows applications to specify only their resource demands, leaving the kernel to satisfy those demands using hidden resource management schemes potentially hidden from the user. This separation of resource specification from resource management allows ffff-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

The resource kernel gets its name from its resource-centricity and its ability to

- apply a uniform resource model for dynamic sharing of different resource types,
- take resource usage specifications from applications,
- guarantee resource allocations at admission time,
- schedule contending activities on a resource based on a well-defined scheme, and
- ensure timeliness by dynamically monitoring and enforcing actual resource usage.

1.3.1 RT-Mach as a Resource Kernel

We have extended RT-Mach[2Y] to serve as a resource kernel[1, 20, 21]. The resource kernel capabilities are obtained by reserving resources for applications requesting them, and tracking outstanding reservation allocations. Based on the timeliness requirements of reservations, the resource kernel prioritizes them, and executes a higher priority reservation before a lower priority reservation *if* both are eligible to execute.

Explicit Resource Parameters: The RT-Mach resource reservation model employs the following parameters: computation time C every T time-units for managing the utilization of a resource, a deadline D for meeting timeliness requirements, a starting time S of the resource allocation, and L, the life-time of the resource allocation. We refer to these parameters, C, T, D, S and L as explicit parameters of our reservation model. The semantics are simple, well-known in the real-time community and are as follows. Each reservation will be allocated C units of usage time every T units of absolute time. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval. The guarantees start at time S and terminate at time $S + L$.

Implicit Resource Parameter] The resource kernel implicitly derives, tracks and enforces an implicit B parameter for each reservation in the system. B represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. If its B is unbounded, a reservation cannot meet its deadline. Priority (or reservation) inheritance is applied when a reservation blocks, waiting for a lower priority reservation to release (say) a lock. As it so happens, this implicit parameter B can be dominant in disk bandwidth management. In communications, B is the transmission time of a single packet, which is generally negligible because of the relatively small maximum size of packets compared to network bandwidth. For example, B in 10 Mbps Ethernet is 1.2 ms corresponding to a Maximum Transmission Unit (**MTU**) of 1 can actually restrict the complexity of scaling switches to a large number of traffic flows to be ff(1). ffhe following techniques based on our understanding of real-time resource management techniques such as ffate-ffonotonic ffnalysis (fixed priority preemptive scheduling and earliest deadline first) will be used to obtain this very practical and interesting result]

- ffe shall associate with each traffic flow only its delay category (of which there can be a geometric progression of 2

Unfortunately, we cannot satisfy this harmonic requirement in the general case. In contrast, earliest deadline scheduling and proportional allocation schemes can utilize 100% of a resource.

The goal of our research is to retain the appealing scalability of the switch while limiting the schedulability loss of the emulated fixed-priority scheme to less than 10%. Our intended output is to define a mathematical relationship between the number of delay categories, the delay values and the loss of schedulability. Fortunately, the approximate 10% loss in schedulable link or switch utilization is not lost completely; that bandwidth can still be used by non-real-time traffic such as `ftp` and `telnet`. In other words, we may only be forced to set aside 10% of throughput for non-real-time traffic, something that would likely be done in practice anyway. Alternatively, the proportional allocation and earliest deadline first schemes use “time” as the base unit, and a fine representation of time requires a large number of bits (32 to Y4), and in addition, a priority queue based on keys of this length may become necessary.

Finally, our proposed research will clearly define the mechanisms for performing admission control and resource enforcement in this environment.

1.5 Organization of the Report

The rest of this report is organized as follows. Chapter 2 discusses the brief description of network protocol processing in two operating systems, FreeBSD and Real-time Mach. Chapter 3 describes the design of our new network packet scheduler and its implementation. Chapter 4 presents the experimental evaluation that explores the performance guarantees in using the reserves. Finally, Chapter

Chapter 2

Network Protocol Processing

In this chapter, we shall discuss the network protocol architecture of two operating systems: FreeBSD and Real-time Mach.

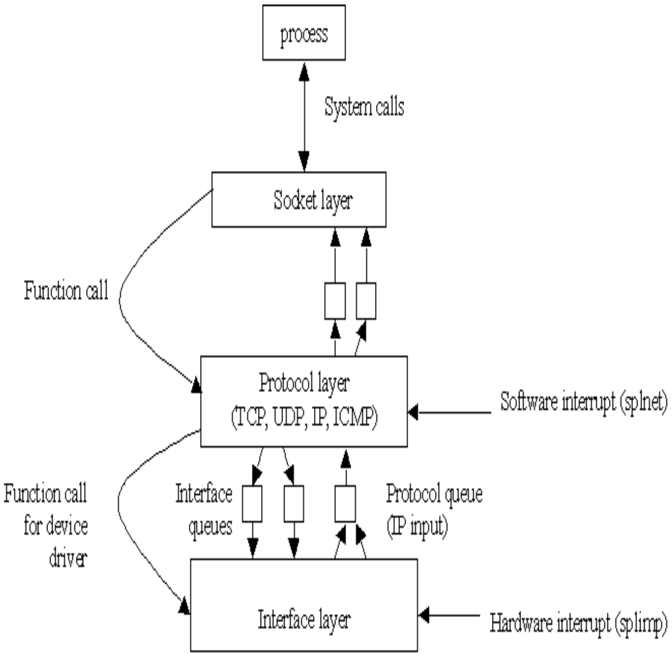
2.1 Network Protocol Processing in FreeBSD

2.1.1 Layered Architecture

FreeBSD is a derivative of the 4.4 BSD operating system. Network protocol processing occurs in mainly three different layers of the kernel: the *interface* layer, the *protocol* layer and the *socket* layer. The *interface* layer contains the device drivers that communicate with the network devices. The *protocol* layer comprises of the network protocol stack. The *socket* layer provides the protocol independent *application program interfaces (APIs)* to the protocol-stack. The applications use network resources through *socket* system calls such as `socket()`, `bind()`, `sendto()`, `recvfrom()`. Figure 2.1 summarizes the communication that takes place among the layers for both the network input and the network output. Memory buffers, known as `mbufs`, are used to hold various pieces of information throughout the networking code, as the user data travels from the process to the network interface, and vice versa.

2.1.2 Output Processing

The network output processing starts when the application makes a system call, e.g., `sendto()` to transmit a UDP datagram. The processing in each layer goes through a series of function calls made by the upper layers. The user data is contained in a linked list of `mbufs`. Each function call passes a pointer to the list to be output. There is no queuing of the data `mbuf` chain until it arrives at the device driver. There is one queue per socket and one queue per interface (Ethernet, SLIP,



ffigure 2.1] ffommunication among network layers for ffreeffffff

loop-back etc.) for an outbound packet.

2.1.3 Input Processing

ffhe input processing being asynchronous, is different from the output processing. ffhe processing in the layers is controlled by hardware interrupts. ffhe handler for hardware interrupt from the network interface device (*splimp*) copies the received packet to the interface layer, into an mbuf chain. ffhe device driver passes the mbuf to a general ffthernet routine which looks at the type field in the ffthernet frame to determine the network protocol (e.g, ffff) of the packet. ffhen, it adds the mbuf to the input queue of the network layer, and generates a software interrupt (*splnet*) that will cause the protocol layer to be executed. ffhere is one input queue per protocol. ffor example, there is only one ffff input queue.

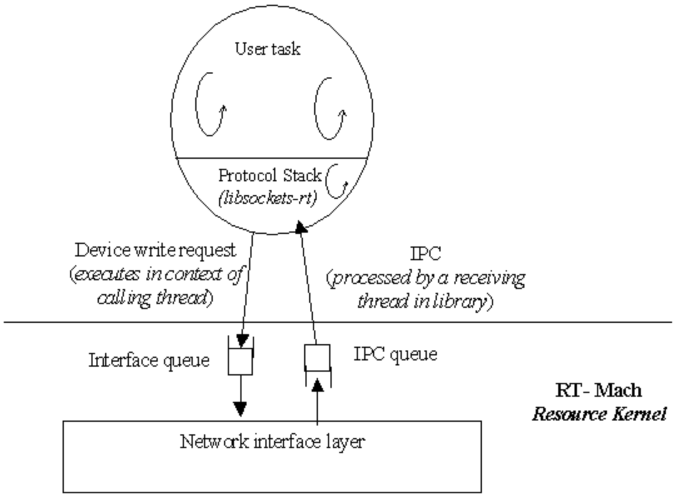


Figure 2.2: Network architecture for Real-time Mach

2.2 Network Protocol Processing in RT-Mach

2.2.1 Architecture

RT-Mach is a micro-kernel based operating system. Networking in RT-Mach is accomplished by calling the 4.4 BSD-Lite networking primitives which are handled by the 4.4 BSD-Lite server. The server can interact with the network device driver residing in the kernel to send and receive packets. But, to improve throughput and latency, RT-Mach also has a library implementation of the protocol stack named *libsockets-rt*[14]. Thus the application task directly interacts with the network interface in the kernel through the socket library. This is illustrated in Figure 2.2.

2.2.2 Output Processing

The protocol stack passes the `mbuf` chain to the network device in the kernel using '*device write request*'. Thus the interface layer in the kernel receives the data in the form of a '*io request*' chain. It puts the '*io request*' packet in the queue and calls the device driver.

2.2.3 Input Processing

ffn case of input processing, the network interface layer receives the packet from the device driver,puts it in the message queue for ffffff. ffhe application receives the packet through network thread in the socket library.

2.3 Summary

ffhis chapter provided an overview of the network protocol processing mechanisms of ffreeffffffff24ff and ffeal-time ffach.

Chapter 3

Scheduling Network Packets

In this chapter, we shall discuss our network bandwidth reservation scheme. Each application specifies its network bandwidth requirements in a 3-tuple (C, T, r), where C is the volume of data in terms of number of bits to be sent/received over a recurring time period T, and r is the type of reservation that determines how the reservation behaves upon getting depleted. First, we briefly discuss the implementation of the traditional packet scheduler in an Ethernet device driver. The reservation types we support will be presented next. Then, we will discuss the design of admission control and scheduling algorithms and its implementations. These modules involve changes in the device driver and the interface layer of the network protocol stack.

3.1 Packet Scheduling in the Device Driver

As discussed earlier, we have one *send* queue associated with each network interface. A packet gets queued after being encapsulated by the Ethernet header. The device driver calls '`start`' routine [1] that dequeues these packets and arranges them to be transmitted by the network interface card. It keeps dequeuing the packets until the queue is empty or unless it gets interrupted by the driver interrupt on the reception of a packet from the network. Similar to alternate queuing support ff2Zff for FreeBSD, we modified the queuing interface and associated multiple *send* queues of different priorities with each network interface in our proposed scheme.

On receiving a packet, the device generates an interrupt. The device driver '`read`' [2] routine copies the packet from the device, and delivers it to `ether input` for further processing. The

---

[1] For example, `vxstart` routine is the "start" routine for the 3com 3C59x.

[2] Similarly, `vxread` is the "read" routine for the 3com 3C59x.

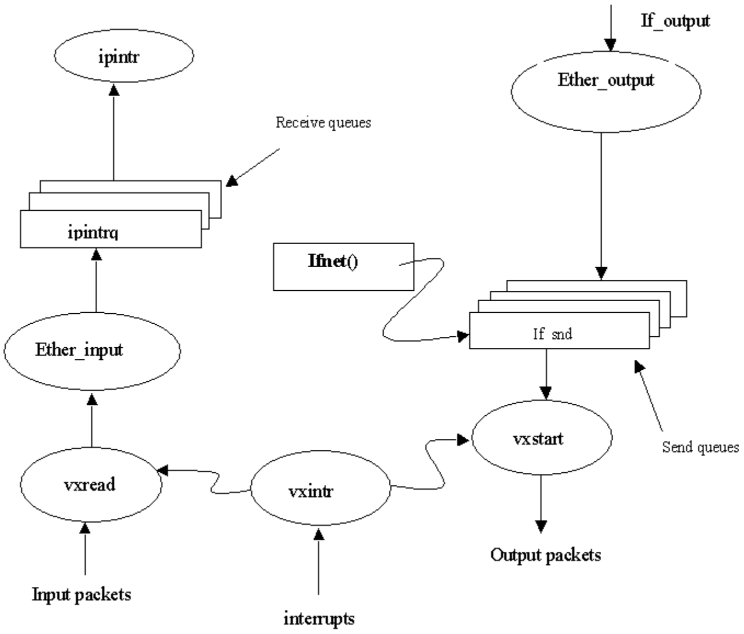


Figure 3.1] Ethernet device driver

`ether input` function queues the packet to the IP protocol queue[3]. In our scheme, we designed multiple queues of different priorities associated with the IP layer.

The entire device driver mechanism is shown in Figure 3.1.

3.2 Reservation Type

As per the resource kernel model, we define three different types of reservations, associated with the parameter r.

- A reservation is said to be *hard reserved* if it is not allowed to use more than its guaranteed network resource.
- A reservation is said be *firm reserved* if it can get more than its allocated resource if no other

---

[3] We are only concerned about the Internet protocol here.

reserved or unreserved applications are ready to use the network.

- Our current implementation does not support *soft reservations*.

3.3 Admission Control

The kernel receives the reservation request from the application in a 3-tuple (C, T, r), where (C, T, r) were defined at the beginning of this chapter. On the sending side, the kernel assigns a $flowID$ to that reservation and also assigns a priority to the queue associated with it. On the receiving side, the application requests a reservation from a specific host with a $flowID$. The kernel reserves the queue for that particular flow. Along with the reservation, the application can also request a certain amount of buffer space. The priority assignment currently follows the *rate-monotonic* fixed priority algorithm [22]. Hence the task with lower value T gets higher priority.

For this scheme, if we assume a point-to-point network where each node has its own bandwidth, the transmission time of a reservation τ_i is dependent on that bandwidth of the host and is given by,

$$S_i = C_i/B$$

where B = link bandwidth.

Under this assumption, the response times for a set of n reservations $\tau_1, \tau_2, ..., \tau_n$ with priority$(\tau_i) < \tau_{i+1}$ can be calculated as follows [20],

$$\forall i, 1 \leq i \leq n, \left(\sum_{j=1}^{i} \frac{S_j}{T_i} \left\lceil \frac{T_i}{T_j} \right\rceil \right) \leq 1$$

In algorithmic form, the response time RT for each reservation can be calculated as follows [20]

1. Set $w_i^0 := S_i$.
2. Compute $w_i^{k+1} := \sum_{j=1}^{i-1} S_j \left(\left\lceil \frac{w_i^k}{T_j} \right\rceil \right)$
3. If $u_i^{k+1} > T_i$, $RT := \infty$, goto step Y.
4. If $u_i^{k+1} = w_i^k$, $RT := w_i^{k+1}$, goto step Y.

The resource kernel will deny the newest incoming request if the response time test fails for at least one reservation.

3.4 Network Packet scheduling and reservation in FreeBSD and Real-time Mach

3.4.1 Scheduling Mechanism

Each reservation is assigned a unique flow identification number $flowID$ that maps to a dedicated priority queue, and is associated with the parameters (C, T, c, t), where t is the elapsed time during a period and c is the volume of data (in bits, for example) that has been transferred/serviced over t. These two parameters get reset at the end of each interval T. The packet scheduling algorithm on the sending side is different from that on the receiving side.

On the sending side, we have multiple priority queues associated with each network interface. The packet scheduler chooses the highest priority queue, checks the status associated with the queue (i.e., c and t), and dequeues a packet if the reservation is not yet exhausted within the time period T. Otherwise, it goes to the next highest priority queue.

On the other hand, we have multiple priority queues on the IP layer for the receiving side. The packet scheduler on the receiving side checks the packet reservation status and puts the packet in the proper protocol queue. The scheduler puts the unreserved packet in the lowest priority queue, and it puts the reserved packet in the queue only if the reservation is not yet completely consumed. Otherwise it drops the packet.

3.4.2 Sending-side scheduling algorithm

On the sending side, we are mainly concerned with the network as a bottleneck. We prevent the network from getting congested by not letting unwanted packets pass through. The routine *GetPacket()* in the network layer receives the packet from the upper layer, puts it in the proper `send` queue. The network device driver chooses the queue based on the priority, reads the header of the first packet, determines the session associated with the packet, checks the reservation and the eligibility status of the session, dequeues the packet and sends it to the network. The eligibility is based on the parameters (c, t), which tells whether the reservation of that session is depleted[4]

---

[4] A reservation is said to be depleted when it uses up its allocation of C within the interval T, i.e., $c = C$ when $t \leq T$.

or not. Finally, after sending the packets, we update the resource parameter (c). The distribution of the excess network resources is done on the basis of priority, and reservation type. In a special case when all the hard-reserved sessions have exhausted their quota, we choose the session with a minimum value of $(T-t)$, and reschedule the device driver start routine after the interval of $(T-t)$ time units. The detailed description of the algorithm is as shown below.

```
Get-Packet()

1. Determine the flow-ID of the packet.

2. Mark the flow-ID 1 if the packet belongs to an unreserved session.

3. Determine the corresponding priority queue and place the packet on
the queue.


Send-Packet()

1. Start with the highest priority queue.

2. Check the parameters (c,t) of the class/reservation associated with
the queue.

3. If the session is either depleted or it is about to exceed its
reservation, mark the flow-ID 'depleted', select the next priority queue,
goto 2.
 else, dequeue the header packet and send it to the network.

4. If all the reservations are depleted, select the highest priority
firm-reserved session.

5. If no firm-reserved session is present, take the parameters (T,t),
compute the (T-t) of all the reserved sessions, and choose the session
with minimum value of (T-t) and schedule the device driver routine
after (T-t) units of time.
```

3.4.3 Receiving-side scheduling algorithm

Packet scheduling on the receiving side is relatively tricky. One would like to deliver packets to the reserved sessions on a timely basis. However, packets must be received and pre-processed to determine which packets belong to reserved session. We want to drop the ineligible packets as soon as possible in order to save the processor cycles for the reserved sessions. A long stream of packets for unreserved sessions can consume fffff cycles that are therefore *not* available for processor applications.

Generally, unreserved sessions have a dedicated low-priority queue and we drop packets if its buffer is full. All the reserved sessions will have their queues of priorities higher than that of the unreserved queue. Any packet belonging to a reserved session can be uniquely identified by its *flowID* and *source IP address*. We describe the algorithm in detail as follows.

```
Receive-Packet()

1. Determine the flow-ID and source IP address of the packet.

2. If the packet belongs to a reserved session,
if the reservation is not yet depleted,
put it in the proper priority queue.
else, drop the packet.

3. Else, put the packet in the unreserved queue.

Dequeue-Packet()

1. Choose the highest priority queue.

2. If the queue is not empty, dequeue a packet.
else , repeat step 2.

3. Return if all the queues are full.
```

It is important to note that this algorithm does *not* take care of the loss in the processor cycles when the host keeps receiving unwanted packets from a potentially malicious application over the network. Therefore, we strongly recommend that the default unreserved session be bound to a fixed limited reserved session that can be changed dynamically. Hence, any unwanted packets can be dropped before being enqueued.

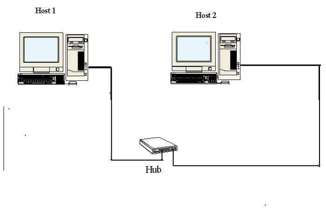
3.5 Summary

In this chapter we described our packet scheduling and admission control control algorithms. We also discussed in brief the implementation of an Ethernet device driver, the associated network protocol stack, and their modifications to incorporate our network bandwidth management scheme.

Chapter 4

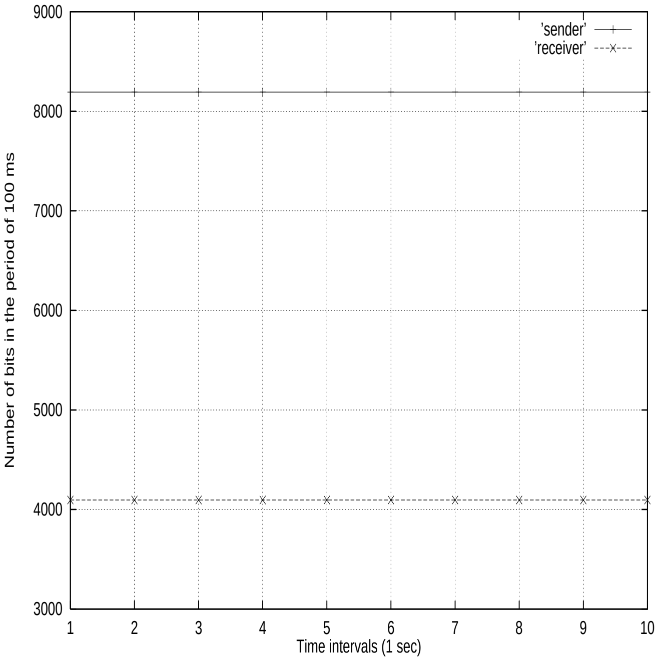
Performance Evaluation

ffe conducted a series of experiments to study the performance of the network packet scheduling algorithm on both ffreeffffff and ffff-ffach. ffe modified ffreeffffff 2.2.1-ffffffffffff and ffeal-time ffach ffff“[b to incorporate our network bandwidth reservation scheme. ffur measurements were carried out on two personal computers, both being ffell dimension fffff 300ff ffentium-ffff 300 ffffz with Y4 ffff memory. ffach of them is equipped with a 3com 3ff“



ffigure 4.1] ffetwork configuration for the experiments

transmitting one ffffff packet between two ffff-ffach workstations using *libsockets-rt* was .4



ffigure 4.2] ffirm reserved sender, with 4000 bits/100 ms reservation on both sides

4.3 Experiments on Network Reserves

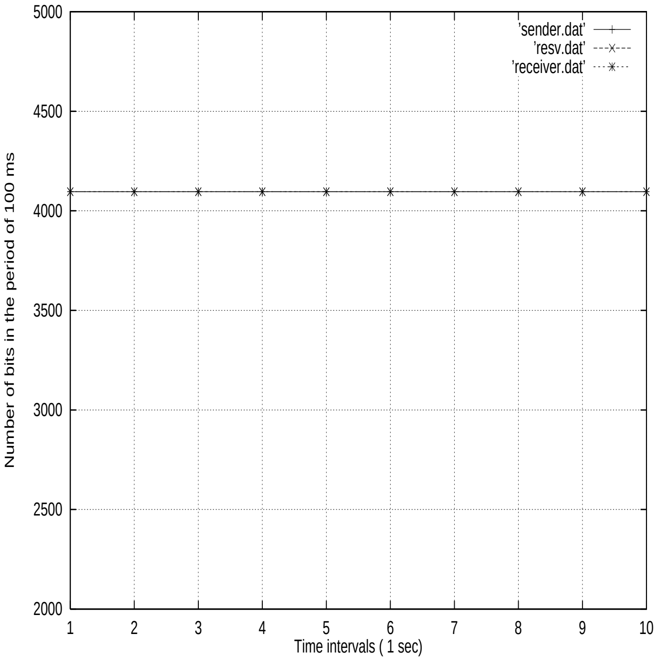
ffe now evaluate the network bandwidth reservation scheme by running different workloads with and without the reservation. ffe illustrate two basic points in these experimentsff viz. the nature of the two types of reservations and the flexibility to upgrade and downgrade a reservation dynamically.

4.3.1 ***Experiment 1 : Reservation Types***

ffn this experiment, we ran two simple applications to illustrate the hard and firm network bandwidth reservation types. ffhey were]

- a sending ffffff agent on ffost 1 that generates traffic at the rate of [ff bits every 100 ms and has a reservation of 4ff bits every 100 ms.
- a receiving application on ffost 2 that captures those ffffff packets and has a reservation of 4ff bits per 100 ms.

fft first, we set the sending side as *firm reserved.* ffe measured the number of bytes that were being sent to the network by the sender application in ffost 1 and the same that were being received by the receiver application on ffost 2 every period of 100 ms. ffhe snapshots of the measurements



ffigure 4.3] ffard reserved sender, with 4000 bits/100 *ms* reservation on both sides

are shown in ffigure 4.2.

ffext, we ran the same applications with the same reservation parameters, setting the sending side reservation to *hard*. ffhe plotted results are shown in ffigure 4.3.

ffhe sending and receiving bandwidths are measured in the interface layer and as expected, an equal number of bits (4ff bits in 100 ms) were being sent/received in the two cases respectively. fflthough the sending side was able to send to its full capacity under firm reservation, it was only able to send 4ff bits every 100 *ms* under a hard reservation. ffhe latter was equal to the number of bits that were being received on the receiving side. ffhis experiment was carried out in both ffreeffffff and ffff-ffach, and identical results were obtained.

4.3.2 *Experiment 2: Reserved Senders and Receivers*

ffhis experiment illustrates that multiple senders and receivers can obtain timely access to network bandwidth.

ffn this experiment, we had *two* hard reserved network applications on ffost 1, with reservations of 100000 bits per 200 *ms* and 1

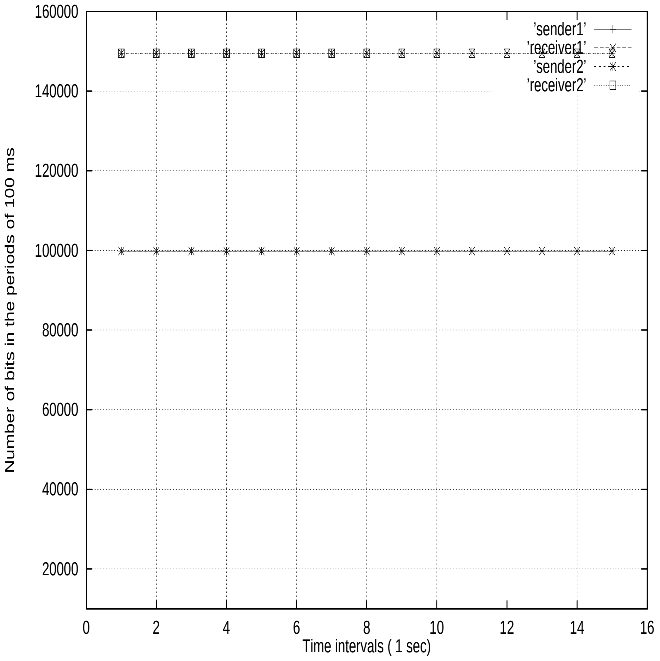


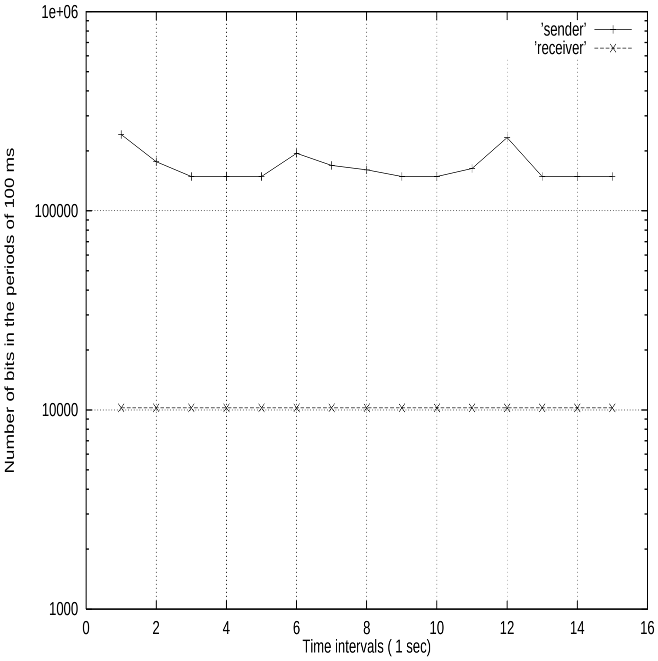
Figure 4.4] Hard reserved senders and receivers

bits equal to the throughput of their respective senders. The result is illustrated in Figure 4.4.

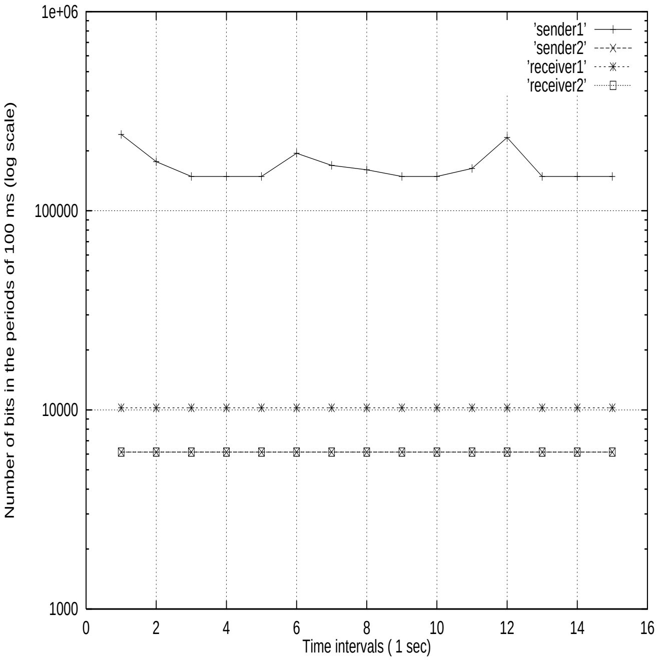
4.3.3 *Experiment 3: Mis-behaving Senders*

This experiment will illustrate that a misbehaving sender cannot force a receiver to handle larger than expected traffic.

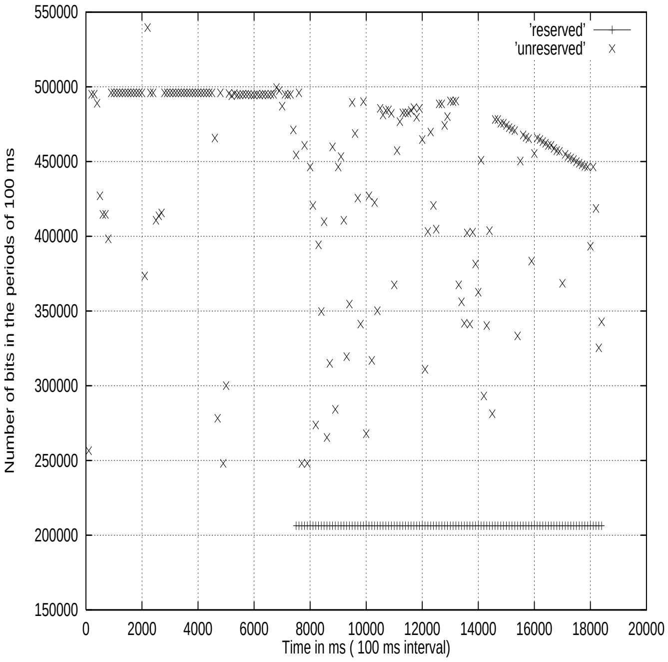
This experiment involved an unreserved sender on Host 1 and a reserved receiver on Host 2. The sender was able to send its maximum possible rate which varies over time whereas the receiver was not able to receive more than its reserved quota of bytes. The receiving side had the reservation of 10ff bits per 1


1e+06
100000
10000
1000
0
2
4
6
8
10
12
14
16
Number of bits in the periods of 100 ms
Time intervals (1 sec)
'sender'
'receiver'

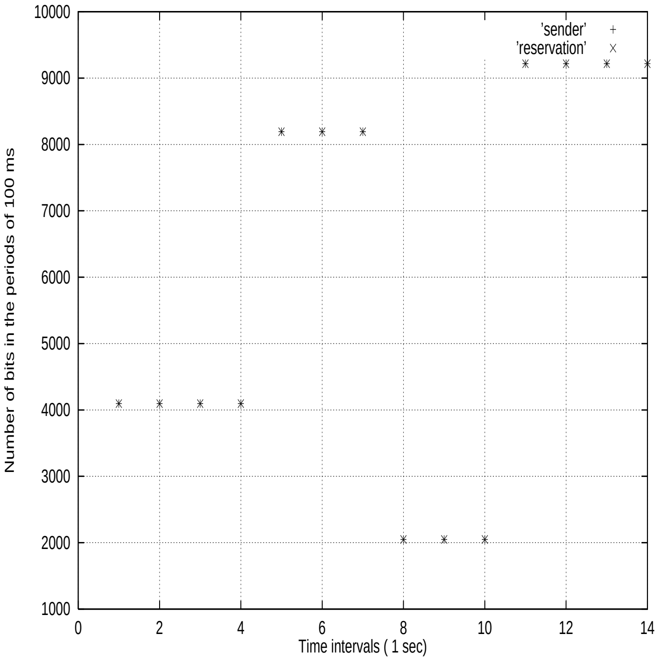
ffigure 4.



ffigure 4.Y] ffne reserved and one unreserved sending applications, reserved receiving applications, (y-axis in log-scale)



ffigure 4.Z] ffne reserved and one unreserved sending applications, reserved receiving applications, (y-axis in log-scale)



ffigure 4.[] ffynamic changes in reservation parameters

4.3.6 *Experiment 6: Dynamic Changes in Reservation Parameters*

ffhis experiment illustrates that the parameters of a reservation can be changed dynamically[2].

ffn this experiment, we had a hard reserved sender whose reservation we dynamically changed from time to time and we observed the corresponding change in its throughput. ffhe sender application started with a reservation of 4ff bits per 100 *ms*ff it got changed to [ff bits per 100 *ms* at the time instant of 4 *s*, and it again got reduced to 2ff bits/100 *ms*, and finally it was raised to “ff bits/100 *ms*. ffhe corresponding network throughputs for the application were 40“Y bits/100 *ms*, [1“2 bits/100 *ms*, 204[bits/100 *ms* and “21Y bits/100 *ms*. ffhe result is shown in ffigure 4.[. ffs mentioned earlier, these numbers correspond to multiples of Y4 bytes, the smallest packet size over the ffthernet.

4.4 Experiment on Integral Management of Processor Reservation and Network Reservation

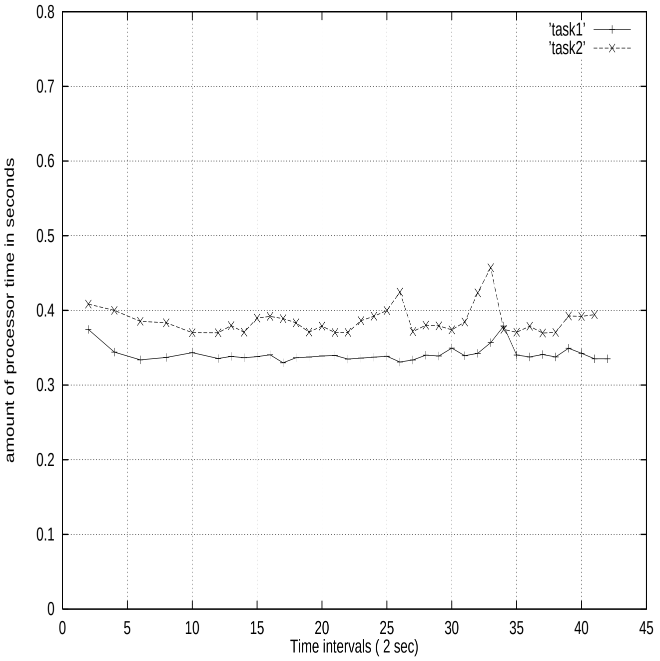
ffe now consider an application which needs both processor and network reservations. ffhis experiment is conducted on the ffff-ffach resource kernel which supports both kinds of reservations.

---

[2] Such dynamic changes may be made by an adaptive QoS manager, for example.

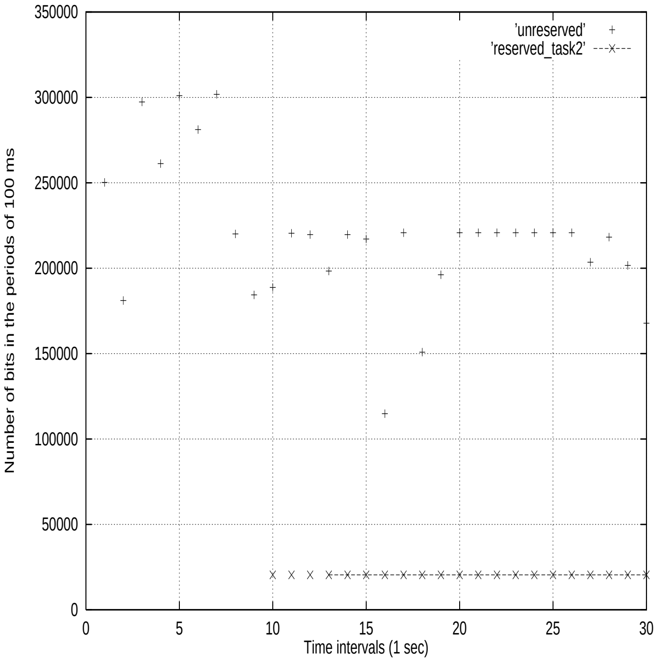
This experiment involved a task with processor reservation (task 1), a task with processor reservation and network reservation (task 2), a non real-time task (task 3) involving processor computations and an unreserved task (task 4) trying to send lot of data across the network using very few processor cycles.

Task 1 had a processor reservation of 2 ms every 10 ms (fraction of the processor time was 0.2). Task 2 had a processor reservation of 11 ms every



ffigure 4.“] ffrocessor reservation for task 1 task 2

consumption dropped to approximately 1.



ffigure 4.10] ffetwork reservation for task 2 in presence of an unreserved task

Chapter 5

Conclusion

We studied the problem of ffff resource management in the context of network bandwidth reservation, where multiple tasks with different timing constraints access multiple resources including the network. In the context of the resource kernel model, we extended the abstraction for operating system reservation using *rate-monotonic scheduling theory* in the domain of networks.

The implementation of network bandwidth reserves requires each task to specify its requirements using a tuple (C, T, r), where C is the volume of data (in number of bits) to be sent/received across the network over a time period T and r is the type of reservation. The request undergoes admission control procedure using a *rate-monotonic* fixed priority algorithm. Unlike the case of proportional fair-share scheduling, this approach does *not* look at the absolute fraction of total bandwidth of the reservesff and assigns priorities based on their time periods (T).

We implemented multiple queues of different priorities in the network interface layer, associated with each reservation of the sending applications. Similarly, we had multiple queues in the ffff layers for the receiving applications. Hard and firm reservation types are supported, while soft reservations are yet to be implemented.

The experimental evaluation demonstrated that the applications using the reservation can achieve predictable timing behavior. The hard reserved applications were able to achieve very consistent real-time performance guarantees on the network, even in the face of competition from other reserved and unreserved applications. In addition, the early-drop policy of the ineligible packets on the receiving side showed and processor management (and, occasionally, storage media management too). Our future work includes the management of multiple resources and distributed resource reservation in deeply networked systems. This includes the integration of the processor reservation within the context of network resource management by the operating system. This also includes the extension of the bandwidth reservation beyond the end-host points, i.e., across network routers and switches. Consequently, the work in this thesis also opens grounds for comparisons between *rate-monotonic analysis* and other scheduling mechanisms ff3, [ff.

Bibliography

ff1ff ff.ff. ffercer *Operating System Resource Reservation for Real-time and Multimedia Applications* ffchool of ffomputer ffcience, ffarnegie ffellon ffniversity ffune “Z

ff2ff ff. ff. ff. ffennett, ff. ff. fftephens and ff. ffhang. *High Speed, Scalable, and Accurate Implementation of Fair Queueing Algorithms in ATM Networks.* fffffff ’“Z.

ff3ff ff. ff. ff. ffennett and ff. ffhang. ffierarchical ffacket ffair ffueueing fflgorithms. *IEEE/ACM Transactions on Networking*, ff“ff ff. ff. ffones and ff. ff. ffeach. ffodular real-time resource management in the rialto operating system. ffechnical ffeport fffff-ffff-“

ff20ff ff. ffajkumar, ff. ffuvva, ff. ffolano and ff. ffikawa. ffesource ffernels] ff ffesource-ffentric ffpproach to ffeal-time and ffultimedia ffperating ffystems ffn *Proceedings of the SPIE Conference on Multimedia Computing and Networking.* ffanuary, 1““[.

ff21ff ff. ffolano, ff. ffuvva and ff. ffajkumar. ffuaranteeing ffiming ffonstraints for ffisk ffccesses in ffff-ffach ffn *Proceedings of the IEEE Real-Time Systems Symposium.*

ff22ff ff. ff. fflein, ff. ffalya, ff. ffollak, ff. ffbenza, and ff. ff. ffarbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate-Monotonic Analysis for Real-Time Systems.* ffluwer ffcademic ffublishers, 1““3. ffffff 0-Z“23-“3Y1-“.

ff23ff ff. fftoica, ff. ffbdel-ffahab, ff. ffeffay, ff. ffaruah, ff. ffehrke and ff. fflaxton. ff ffroportional ffhare ffesource ffllocation fflgorithm for ffeal-ffime ffime-ffhared ffystems. ffn *Proceedings of the 17th IEEE Real-Time Systems Symposium.* pp 2[[-2““, ffec 1““Y.

ff24ff ff. ff. ffright and ff. ff. fftevens. *TCP/IP Illustrated, Volume 2, The Implementation.* ffddison ffesley, ffeading ffff, 1““