Resource Management of the OS Network Subsystem

Sourav Ghosh and Ragunathan (Raj) Rajkumar
Real-time and Multimedia Systems Laboratory
Department of Electrical and Computer Engineering
Carnegie Mellon University
sourav, raj@ece.cmu.edu

Abstract

A QoS-aware real-time operating system must schedule multiple tasks which have different timing constraints and which access various resources including the CPU, disk and network. These resources, however, are not independent of one another. For example, resources like network bandwidth and disk bandwidth are available on a single node but must be managed by their host OS on the CPU by means of interrupt handlers, device drivers, file-systems and/or protocol services. Hence, in order to obtain guaranteed completion times, an application must therefore obtain both user-mode time on the CPU along with sufficient OS-level time for the network and disk subsystems. In this paper, we investigate the co-scheduling of CPU cycles and network bandwidth. Specifically, we study the problem of obtaining pre-specified network bandwidth received by applications from the external network. Our solution endows (1)direct control over the flow of network packets into the system based on the requirements of specific applications, (2)guaranteed and enforced processing time for the received packets,(3)precise accounting of those processing times, and (4)elimination of scheduling anomalies. We describe and evaluate our system design and implementation in Linux/RK, a QoS-aware real-time version of Linux. We also compare this approach with a commercial implementation we did in TimeSys Linux.

1 Introduction

The main goal of resource management in a real-time operating system is to provide timely, guaranteed and protected access to system resources. The resources are in general not completely independent of each other. Hence, there are situations where the guaranteed access of one resource disrupts the same of another, giving rise to scheduling anomalies.

For example, a multimedia application needs to send real-time data across the network and demands a certain amount of network bandwidth reservation and timely response from the system. A traditional network scheduler mainly involves a packet-queuing discipline that performs quick transmission of packets. But the application itself needs to generate enough data in timely fashion in order to supply an adequate number of packets to the network scheduler. This comes at the expense of a significant number of processor cycles both at the user-space and the system-space. Consequently, this calls for processing guarantees for the application and the packet scheduler at the same time.

It must be noted that the task of receiving data from the network is more processor-centric. The system does not have direct control on the number of packets arriving on its network device. But it can certainly control the processing of these packets in order to provide guaranteed service to various applications with different timing constraints.

If we investigate network protocol processing in the standard Linux operating system, it does not provide timing guarantees on the processing of inbound packets. This processing is essentially interrupt-driven. The arrival of a packet in the network device generates a hardware interrupt whose job is to capture and store the packets. This in turn generates a software interrupt (Bottom Half) that performs the protocol processing of packets making them ready for the applications. An application, upon being scheduled to run, retrieves packets from the system-space for its own processing. The whole procedure involves considerable amount of system-level processing that is not very much under the control of the application. In other words, the user-level applications will always be preempted by the system-level activities for processing network packets. Moreover, this processing time in the system space is charged to the preempted process, which might not be the process that will eventually receive those packets.

The persistent handling of incoming network packets at a high priority leads to scheduling anomalies, priority inversion and even overall decreased network throughput. In the extreme case, the packets can arrive at a rate fast enough where the system will spend all of its time on grabbing and storing the packets, and finally dropping them as lost when the queues run out of buffers. Thus, all the inbound data are lost as the application is never scheduled to run, a phenomenon popularly called "receiver-livelock".

1.1 Related Work

We consider the two primary objectives of a real-time operating system design to be

• **Predictability**: To provide predictable performance to the applications.

• **Enforceability**: To bound and control the resources consumed by different activities.

From the network subsystem design perspective, Chen Lee et al.[4] first developed a library-based real-time network protocol processing scheme for the RT-Mach operating system[12]. RT-Mach is a microkernel-based operating system where a small kernel performs a few low-level functions. All other jobs are handled by user-level processes and servers. For example, networking in RT-Mach is accomplished by calling the BSD networking primitives that are handled by a user-level BSD Unix server. This server interacts with the network device driver in the kernel to send and receive packets. But using the socket library, an user-level task interacts directly with the network interface in the kernel and performs protocol processing of its packets.

"The Lazy receiver processing (LRP)" network subsystem architecture [5] by Druschel and Banga addresses the issues of scheduling anomalies and receiver-livelock in a monolithic Unix operating system. The core idea is to incorporate early-demultiplexing of packets based on their destinations and processing them at the application-level priorities. The processing type varies depending on the protocol type of the packets. UDP packets are in general processed synchronously, i.e., in the context of the application issuing the "receive" system call. For TCP packets, LRP creates a kernel-thread for each user-process that receives those packets and charges its resource utilization to the same process. Later, they introduced resource containers [9] as "chargeable entities" where multiple processes can be grouped together. Thus, LRP charges resource usage on processing packets to the corresponding resource container.

Brustoloni et al.[7] identified a few limitations of LRP, and suggested a new scheme known as "Signaled Receiver Processing(SRP)". The main shortcomings of LRP scheme that were cited are: (1) LRP relies on the existence of a kernel thread that is not available in some operating systems, such as FreeBSD; (2) LRP has only one level of demultiplexing, which is not suitable for gateways where packets may need to be demultiplexed more than once; (3) The LRP architecture is not flexible enough to support synchronous TCP processing which is sometimes beneficial to TCP because of better memory locality; (4) it does not clearly explain how this scheme will work with a real-time scheduler.

SRP comprises of the following features. Firstly, it has multiple stages of demultiplexing that is suitable for gateways. Secondly, it uses a special signal to notify the socket's owner processes. This default signal handler dequeues the packets and submits to its respective sockets' input function.

Our work is most closely related to that of Jeffay et al. [6]. It involved real-time scheduling of both the user-level and system-level activities based on a proportional-share scheduler in a general-purpose monolithic operating system. The system-activities are scheduled along with the user-level processes. On scheduling system activities, their work primarily focused on scheduling the network protocol stack for the received network packets.

Our objective is to provide guaranteed and timely access to system resources for real-time applications by means of integrated (or cooperative) scheduling of multiple resources. We focus on the real-time scheduling of network subsystem that involves integrated scheduling of network bandwidth and processor cycles. We perform our work in the context of Carnegie Mellon's Resource Kernel [1] approach, which will be described briefly next.

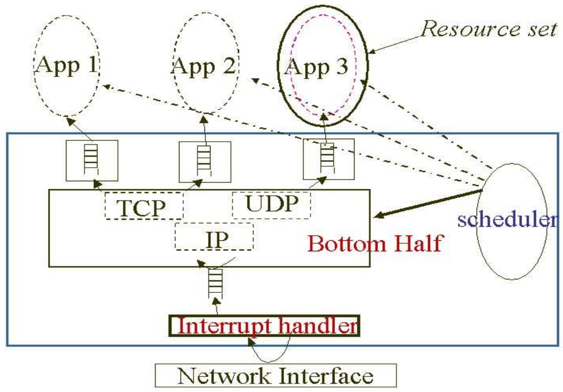

Figure 1: The Network Subsystem in Linux

1.2 Resource Kernel

A *resource kernel* [1] is a resource-centric approach for building real-time kernels that provides timely, guaranteed and enforced access to system resources. A resource kernel allows applications to specify their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. The key features are to:

•apply a uniform resource model for dynamic sharing of different resource types,

•guarantee resource allocations at admission time,

•ensure precise timing guarantees and temporal protection between applications by means of well-defined resource usage model on time-multiplexed resources.

The resource kernel concept was first tightly integrated into the RT-Mach operating system and allows guaranteed, timely and enforced access by applications to processor cycles [12], disk bandwidth[2] and network bandwidth[8].

1.2.1 Linux/RK: Resource Kernel in Linux

Shui and Rajkumar [3] developed the "portable resource kernel" i.e., a resource kernel that can be ported to any operating system. This was implemented on a general-purpose operating system Linux, which became popular as "Linux/RK". It comprises of a portable RK subsystem that is nearly independent of any operating systems specifications. TimeSys Corporation, the developer of TimeSys Linux, also derived its initial notions from this work.

"RK" introduced two basic abstractions on resource management in the resource kernel. They are:

1.2.2 Reserve

A *reserve* represents a single share of a computing resource that may be used by the processes in the system. Such resources can be processor time, memory, network bandwidth, or disk bandwidth.

1.2.3 Resource set

A *resource set* represents a set of reserves of various resource types. One or more processes are attached to a resource set, which provides the exclusive use of its reserved amounts of resources shared among those processes.[1]

[1] Similar to "resource containers" in [9].

1.3 Organization of the paper

The rest of the paper is organized as follows. Section 2 describes the problems with the protocol processing mechanism in standard Linux. Section 3 presents the design and implementation of our real-time network subsystem and provides comparisons with the previous schemes. Section 4 discusses its performance evaluation. Finally, Section 5 summarizes our contributions and presents suggestions for future work.

2 Linux Subsystem: Limitations

In this section, we briefly describe the limitations associated with the standard Linux network subsystem.

•The processing of incoming packets at the interrupt handler is considered to be the highest priority activity next to the system clock interrupt. This was based on the historical fact that the peripheral devices used to be much slower compared to the processor speed and packets are only occasionally received from the network. But modern high-speed networks can generate packets at a rate that the system will spend all of its processor cycles just to capture and process a *subset* of those packets, leaving user-level applications completely starved for any resources, and resulting in eventual reduction of network throughput.

•Scheduling anomalies occur as the time spent in the interrupt handler and the bottom half are charged to the application process that was preempted as network packets arrive. Thus, a CPU-bound process can potentially be charged for the network activity in the system.

•There is no scope for prioritized processing of the received packets as all of them share a single backlog queue at the bottom half. The processing of the previously received packets will be delayed by freshly arriving incoming packets since the interrupt handler runs at a higher priority than the bottom half. In addition, an application may also like to treat different network connections differently. For example, this is applicable to a QoS-aware web-server serving multiple network connections and would like to provide different QoS to different network interfaces.

•An application does not have much control on the processing of its received packets in three ways, viz., allocation of processor cycles to perform protocol processing, receiving bandwidth at the application layer independent of the incoming bandwidth at the network device and allocation of memory or buffer-space at the backlog queue.

•Lastly, the process of capturing the received packets from the network device takes precedence over sending the queued packets to the network at the device driver layer. In other words, there is no control infrastructure to enforce the resource usage on sending and receiving mechanisms. Therefore, if a machine has multiple network devices or links connected to different sub-nets (e.g. a software router), it may not be able to send packets through one free link if another link is very busy in capturing and processing received packets.

3 The Real-time Subsystem

This section describes two implementations of the real-time network subsystem based on Linux/RK [3]. At first, we present our work at Carnegie Mellon University. Next, we also discuss and compare our implementation at TimeSys Corporation (www.timesys.com).

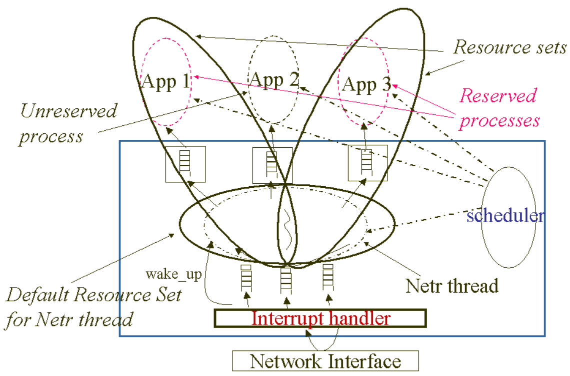

Figure 2: Network Subsystem in Linux/RK

3.1 Linux/RK at Carnegie Mellon

We name our real-time network subsystem as the "`NetR`" subsystem. Figure 2 shows its architecture for Linux/RK. It comprises mainly of the following components:

NetR reserve: Similar to CPU(processor) reserve[11] and network bandwidth reserve[8], we introduce a new reserve known as "receiving network reserve" or `netR` reserve. Each `netR` reserve represents a share of a computing resource, which dictates the rules on how the received packets belonging to a particular reserve should be processed.

The main parameters of a *netR* reserve are:

• `C`: This denotes the volume of data in terms of number of bytes to be received or processed in each period;

• `T`: This is the time duration of the period;

• `D`: This is the deadline in each period within which the processing of the reserved number of bytes should be finished; can be less than or equal to the period T;

• `buffer_space`: Each netR reserve possesses its own dedicated backlog queue. This parameter represents the maximum capacity of the queue.

A `netR` reservation requires a prior creation of a CPU reservation under the same resource set. The protocol processing time for network packets gets charged to the *CPU* reservation. The `netR` reserve dictates the number of bytes of data to be processed in a given time.

Early Demultiplexing: The single global Linux backlog queue is replaced by one backlog queue for *each* valid reservation and a default backlog queue for all packets with no reservation. The network interrupt handler is also modified to demultiplex the incoming packets based on their `netR` reserve, and place them on the appropriate backlog queue.

NetR thread: This is a new kernel thread that replaces the network bottom half and is thereby dedicated to execute protocol processing of arriving network packets. The interrupt handler on receiving a packet wakes up the `NetR thread` instead of activating the bottom half. The `NetR thread` handles packets of different reservations using *Deadline-Monotonic* priorities, which assigns higher priorities to reservations with shorter deadlines (values of D). The unreserved packets at the default backlog queue have the lowest

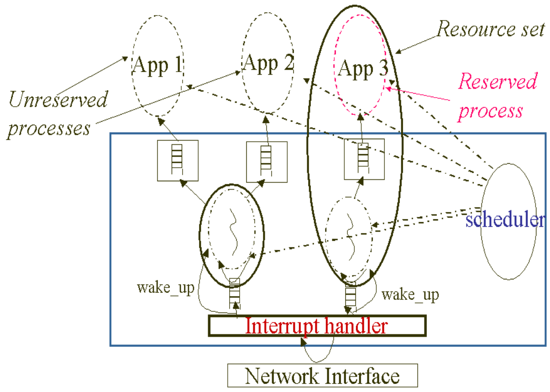

Figure 3: Network Subsystem in TimeSys Linux

processing priority. This thread is scheduled by the CPU scheduler along with other threads and processes.

While processing packets for a particular `netR` reserve, this thread dynamically attaches itself to the corresponding CPU reserve of the same resource set. Thus, the `NetR` thread and the application process *may* share the CPU reserve of a single resource set and the packets are processed in the context of the appropriate resource set. The `NetR` thread by itself can be assigned a certain priority or a CPU reserve by default in order to provide desired performance guarantees to the unreserved packets, as in the case of standard Linux. But the user or the system administrator possesses the power to limit the protocol processing time of the unreserved packets so that extra CPU cycles could be utilized in other more "important" or "real-time" tasks in the system. Since the standard Linux kernel is non-preemptive, we made this kernel thread preemptive at packet processing boundaries.

3.2 TimeSys Linux Architecture

The architecture described now was developed at TimeSys Corporation. It is based on a fully preemptive Linux kernel released as TimeSys Linux 2.0. It consists of all the interrupt handlers (IRQs) and the bottom half running as preemptive kernel threads.

The `netR` reserve implemented in TimeSys Linux differs from the implementation at Carnegie Mellon in some distinct ways. Firstly, the creation of each `netR` reserve spawns a new dedicated kernel thread. This thread performs the protocol processing job pertaining to the particular reserve. It derives its processor cycles from the CPU reserve of the same resource set. Secondly, there is also a generic `NetR` kernel thread that takes care of the processing of the unreserved packets. Thus, we are able to completely isolate the protocol processing jobs of different reservations.

This subsystem also supports synchronous and asynchronous protocol processing requirements. The application process may share the CPU reserve with its `NetR` thread, and a resource set may have more than one process or thread attached to it. In the RK implementation, the threads belonging to the same resource set are contained in a linked list known as `task-list`. When a reservation is eligible to execute, the RK scheduler picks one `runnable` thread that is ahead in the `task-list` with respect to the other `runnable` threads. Therefore, a thread that is sleeping or being blocked on an I/O operation relinquishes CPU control so that another thread in the list can be scheduled. For example, if an application requires "synchronous" protocol processing, it will be put ahead of the `NetR` thread in the list. Thus, the `NetR` thread runs when the application blocks on a `receive` system call, processes a certain amount of data as required by the application and goes to sleep. On the other hand, "asynchronous" protocol processing is accomplished by putting the `NetR` thread at the top of the `task-list`. Under this condition, the `NetR` thread, whenever eligible to run, processes packets until the reserved number of bytes is processed or the backlog queue is empty. When it goes back to sleep, the application thread, on being scheduled to run, makes the `receive` system call to retrieve the processed data from the system.

3.3 Summary of Parameters

The parameters that can be controlled by a user of our `NetR` subsystem are summarized below for the sake of convenience:

- CPU reservation: This is used by the `NetR` thread for processing reserved packets. This may also be used by one or more application processes at the same time.
- `NetR` reservation: This dictates how the received packets for a reserved flow should be processed, given they have enough CPU cycles reserved.
- `NetR` thread priority: The priority of the `NetR` thread determines how "unreserved" packets will be processed. This is *normally* set below the priorities of any CPU-reserved task.
- socket-buffer length: The size of the socket-buffer can be modified as a socket option using `setsockopt`.
- IRQ priorities: Applicable to only TimeSys Linux, the priorities of the IRQ threads and the bottom half threads can be modified. They are normally kept below that of any CPU-reserved task.

3.4 Comparison With Related Work

We now revisit Section 1.1 to precisely compare our approach with other related work.

Our work in a monolithic operating system Linux is analogous to that of `libsockets-rt` in microkernel-based RT-Mach. Unlike the user-level `libsockets-rt` threads in RT-Mach, we have kernel-level `NetR` threads in Linux/RK. However, this avoids the throughput penalty associated with user-level protocol processing.

The use of a kernel thread for protocol processing was also employed for TCP procesing in the LRP architecture. In addition, there is a low-priority kernel thread that processes packets when the system is otherwise idle and the application is blocked on some other I/O activity. Our implementation does *not* require this low-priority kernel thread. In addition, support for synchronous and asynchronous processing is no longer dependent on the type of transport protocol we use. Also unlike LRP, our scheme does not necessarily create a kernel thread for each single connection unless each connection is assigned a different reservation. In other words, multiple connections can share a single reservation. Most importantly, the application has the control on how many packets should be processed in well-defined and repeatable intervals, and how many buffers should be allocated to hold those packets at different levels of processing.

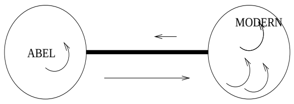

Figure 4: First Experimental Setup

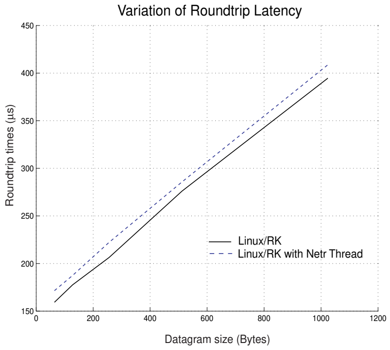

Figure 5: Roundtrip Latency Variation

SRP replaces the use of kernel threads by signals. But the features that kernel threads offer cannot be fulfilled completely by the use of signals. For instance, the signal generated by the incoming packets can only be used to identify the corresponding receiver process and its signal handler. Hence, multiple network connections of different priorities attached with a single process cannot be treated differently using signals. In other words, signals as currently defined cannot scale to support multiple connections on a single process.

Lastly, Jeffay's proportional-share scheduling scheme needs very fine scheduling granularity for better results which in turn significantly increases the context switching overhead within an OS. In that scheme, the hardware and software interrupts are scheduled along with the user-processes. The time quanta for these activities are determined by the number of packets they need to process. But the processing time of a packet varies depending on its protocol type (e.g., UDP, TCP). Hence, it is rather difficult to calibrate these times. In our scheme, applications or aggregates can specify protocol processing time without the need for (uniform and pessimistic) OS calibration. Finally, our scheme also presents explicit user-level control of buffer allocation.

4 Experimental Results

In this section, we perform detailed comparisons of our scheme with the standard Linux scheme. The results were taken using 300 MHz processors and 100Mbps Ethernet. The quantitative aspects of our results do depend on the processor frequency and the network bandwidth. But the need of the `NetR` subsystem will only increase for future systems as network bandwidth keeps going up faster than processor speeds. For example, a typical modern PC has a processor frequency of 1GHz, and the Ethernet bandwidth is 1Gbps. Thus, the ratio of processor frequency to the Ethernet bandwidth is 1, as opposed to 3 in our experimental setup.

The first two experiments compared the raw network performances of Linux/RK with and without our scheme. In other words, they showed how much raw performance is sacrificed in order to provide guaranteed performance.

System	TCP(Mbps)	UDP(Mbps)
Linux/RK	63.19	91.37
Linux/RK with NetR	62.81	91.37

Table 1: Sending-side Throughput

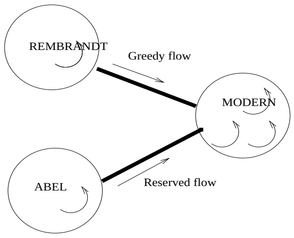

Figure 6: Second Experimental Setup

4.1 UDP Latency Measurements

The first experiment measured the roundtrip UDP latencies under various datagram lengths.

For this experiment, we connected two PCs Abel and Modern running Linux(2.2.14)/RK via a 100Mbps Ethernet hub. The application running on Abel sent a datagram to another application on Modern and waited for the reply. The application on Modern sent the same message back. The hosts were idle otherwise. We measured the time difference for the application running on Abel between sending the datagram and receiving the reply, averaged it over hundred entries and plotted it against the size of the datagram. It measured the roundtrip time between two PCs across the 100Mbps Ethernet as shown in Figure 5.

The plot shows a certain amount of latency penalty associated with the use of `NetR thread` which is independent of the datagram size (the *worst* case being less than 8%).

4.2 Throughput Measurements

This experiment measured the *sending-side* TCP and UDP throughput. The same experimental configuration was retained. The `send` and `receive` socket buffers were kept fixed at 64 KB. We used the `ttcp` [15] benchmark tool for this experiment. Table 1 presents the results.

Since the `NetR` Subsystem only works with the *receiving* packets, the sending-side UDP throughput remained unaffected in both the cases. But TCP transmission relies on received ACKs. Therefore, we observed only a slight dropping of TCP throughput under the `NetR` subsystem.

The results of these first two experiments demonstrate that the basic performance of the `NetR` architecture is comparable to that of normal Linux. In other words, it does not significantly compromise on low-load performance in order to provide guaranteed service and stability under high load.

4.3 Receiving Throughputs

Our next few experiments demonstrate protocol processing guarantees with or without contention using the `NetR` subsystem. As mentioned above, the guaranteed throughput on the receiving packets depends on the following; size of the backlog queue at the interrupt level, *timely* protocol-processing of the received data and the size of the socket-buffer. We will see how these parameters affect the quality of service.

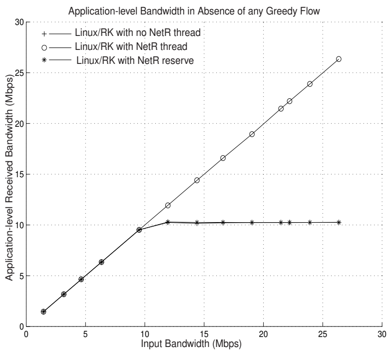

Figure 7: Received bandwidth with no contention

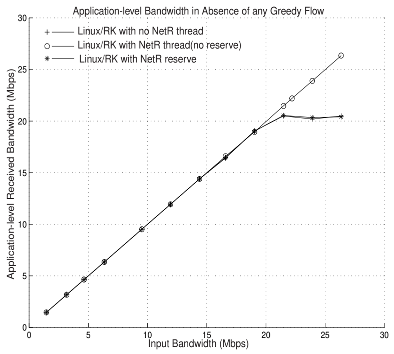

Figure 8: Received bandwidth under larger receive buffers

4.3.1 Throughput with no contention

We kept the same experimental setup. This time, a sender process at Abel sent fixed-sized UDP packets (80 bytes) to a receiver process at Modern. The sending data rate from Abel was again varied by means of a `network` reservation on Linux/RK. The reservation had a period of 4ms. The receiving application received packets using *blocking socket I/O* and discarded them immediately. It was assigned a *Soft* CPU [2] reservation that was more than enough to receive and process data at a certain rate. Figure 7 plots the rate at which the packets were received by the receiver process (in Mbps) against the sending bandwidth of the sender process (in Mbps). According to the figure, the application-level received bandwidth equaled the transmission rate up to a transmission bandwidth of 9 Mbps on standard Linux, beyond which it flattened out. It remained constant up to the measured sending rate of approximately 26 Mbps. This shows that beyond the bandwidth of 10Mbps, buffer-overflow caused loss of data. Since the buffer-space of the backlog queue is larger than that of the socket queue by default, we conjecture that the overflow occurred at the socket queue. This happened since the network bottom half, whose job is to take packets out of the backlog queue, process them and put them at the socket queue, ran too often compared to the application process at high input bandwidth.

Next, the same experiment was performed with the `NetR` subsystem. The `NetR` thread was assigned a priority *lower* than any CPU-reserved thread but higher than other time-shared threads. With the `NetR` thread, the application-level throughput remained at par with the sending rate throughout the measurement range.

Using *blocking socket I/O*, the application blocked on the `recvfrom` system call. And as the protocol processing by the `NetR` thread took place at a lower priority than the CPU-reserved application, it could only run when the application waited to receive data and hence could not cause the socket queue to overflow. This explains why running protocol processing at a lower priority than the application process increases throughput under blocking socket I/O. This is not necessarily true for a few specific applications that perform *non-blocking socket I/O*. A non-blocking I/O inherently assumes that the I/O process runs at a higher priority and therefore it will preempt the application process.

Next, the receiver created a `netR` reservation with the following specifications: parameters C, T, D were chosen based on the specifications at the sending side network reservation at Abel; the *buffer_space* parameter was assigned to be 40% of the total buffer space reserved for received packets (i.e., the backlog queue in standard Linux) in the system. This reserve was attached to the application process so that the `NetR` thread could perform protocol processing at the context of the CPU reserve of the application. As observed from Figure 7, the received throughput under this condition followed that of the standard Linux subsystem. In this case, when the `NetR` thread got dynamically attached to the same resource set as that of the application, it not only inherited its reservation but also got ahead of the application thread in the "`task-list`" queue of the resource set. Therefore, it was scheduled to run before the application causing the same phenomenon of socket-queue overflow as observed in standard Linux. Suppose the `netR` reserve had not been attached to the application process directly, rather only to the sockets involved in the communication so that a separate CPU reserve could be used for the `NetR` thread. Then we would have seen similar throughput as in the second case (`NetR` thread with no `netR` reserve) if that CPU reserve ran at a lower priority than the CPU reserve of the application process.

The experiment shows that running protocol processing at a higher priority than the user-level process is not always desirable for higher throughput. In other words, the relative priorities of the application process and the protocol processing can be modified in our system depending on the application requirements.

In order to verify that overflow at the socket queue indeed caused the drop in throughput, we doubled the socket queue capacity from 64 KB to 128 KB (using the `setsockopt` system call). Then we repeated the same experiment[3]. As shown in Figure 8, the received bandwidth under standard Linux and that under `netR` reserve kept up with the sending bandwidth all the way up to 20Mbps this time, twice that of the previous case (Figure 7).

We kept the socket queue capacity at 128KB for our next set of experiments.

[2]Soft reservation: A reservation is said to be depleted if it has already consumed the amount of resource it reserved. A *Soft* reserved task, upon depletion of its reservation, competes with best-effort tasks to gain access to the excess resource.

[3]The socket option `setsockopt` in a standard operating system such as Linux does not allow the socket queue to increase beyond 128KB by default.

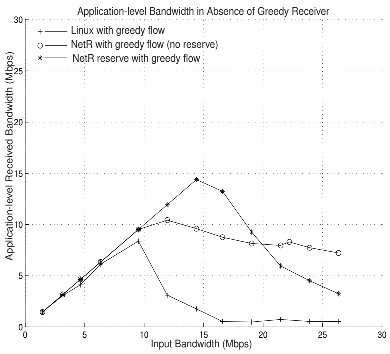

Figure 9: Received bandwidth with greedy sender

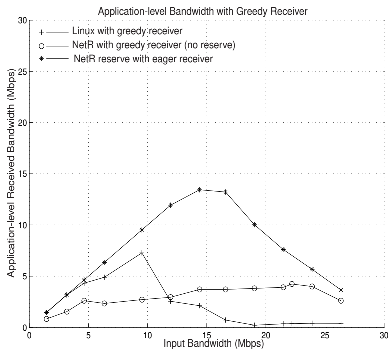

Figure 10: Received bandwidth with greedy receiver

4.3.2 Throughput with greedy contention

This experiment shows how our system provides QoS guarantees to a reserved flow in the presence of a greedy contention.

We changed our experimental setup to that shown in Figure 6. It comprised of three hosts Abel, Modern and Rembrandt. Host Rembrandt had the same specifications as that of Host Abel and Host Modern. Host Modern was connected to Host Abel and Host Rembrandt through two different ports of its multiport network adapter. By isolating the two connections at the physical layer, we avoided any collision or contention for physical network bandwidth. This allowed us to focus only on the CPU processing attributes of the receiving network packets.

Next, we created a single application on Host Rembrandt which kept sending UDP data of very small size (14 bytes) to Host Modern. In other words, we generated a greedy flow from Host Rembrandt to Host Modern that contended for the processing bandwidth at Modern with the reserved flow.

Note that there is no receiver process for this greedy flow at Host Modern. Hence, all packets from Host Rembrandt were eventually discarded. The previous experiment of Section 4.3.1 was then repeated (Figure 9).

As we see, the received throughput under standard Linux failed to keep up with the sending bandwidth much earlier and started going down rapidly at around 9Mbps of the sending rate. Beyond 12Mbps, the throughput went down to a negligible amount because the system spent all of its CPU-time just to capture and process packets mostly from Host Rembrandt and did not let the application run. In other words, *"receiver-livelock"* [10] kicked in.

In contrast, the `NetR` subsystem did early demultiplexing and early dropping of unwanted packets. Thus, the packets from the greedy flow at Rembrandt were discarded at the interrupt handler. The receiver throughput kept up with the sending rate up to 10Mbps and started reducing slowly at around 12Mbps.

With the inclusion of the `netR` reservation, however, the received bandwidth kept pace with the sending bandwidth all the way up to 15Mbps. Hence the system provided the best performance under `netR` reserve. The reason for the sharp drop beyond 15Mbps is mainly due to the uncontrolled activity of the interrupt handler on receiving packets that was charged to the running resource set, and in our case it was the resource set of the receiving application. Comparatively, the degradation was much slower when the `netR` reserve was not present since the resource set of the `NetR` thread and that of the application were different, and they shared the cost of CPU usage of the interrupt handler.

This set of experiments demonstrates that the presence of a `netR` reservation improves the receiving throughput over 50% compared to that of the standard Linux.

4.3.3 Throughput with greedy receiver

This experiment examines how our system works if another receiving application contends for network bandwidth.

In our third setup, a process was created on Host Modern that received packets from the greedy sender at Rembrandt just to discard it. We call it a greedy receiver or an eager receiver. This disabled early dropping of the packets in the `NetR` subsystem from the greedy flow at the interrupt handler. According to the results shown in Figure 10, the performance of standard Linux remained the same as in the absence of the greedy receiver, which was to be expected. This is because standard Linux being devoid of the early demultiplexing feature spent an almost equal amount of time on packets from Host Rembrandt in either case.

On the other hand, the throughput of the `NetR` subsystem with no reservation was very low from the beginning itself, primarily due to buffer overflow at the backlog queue that was shared between the two flows. And this buffer overflow occurs earlier than in standard Linux only because the `NetR` thread ran at a lower priority than the user-level CPU-reserved process and did not get as much CPU as the network bottom half in Linux did to process packets from the backlog queue. Finally, the `NetR` subsystem with `netR` reservation maintained almost the same level of throughput as in the previous case. This is again because the `NetR` thread processed the unreserved packets from Host Rembrandt under its default priority and the reserved packets from Host Modern under the CPU reservation of the corresponding receiver process. In addition, the unreserved and the reserved packets got demultiplexed to different queues at the interrupt handler.

The above two experiments demonstrate the ability of the `netR` reservation to sustain a higher throughput in presence of a greedy contention. The reason for its drop beyond 15 Mbps, as mentioned above, is due to uncontrolled and uncounted activities of the interrupt handler. We shall return to this in the next section.

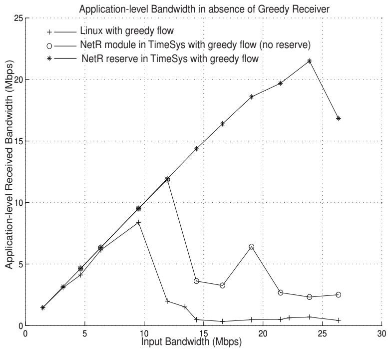

Figure 11: Received bandwidth with greedy flow in TimeSys Linux

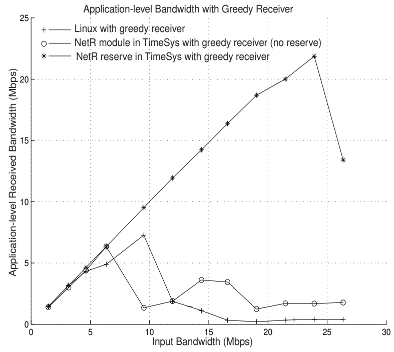

Figure 12: Received bandwidth with greedy receiver for TimeSys Linux

4.4 Throughput in TimeSys Linux

Our next set of experiments deals with TimeSys Linux and its `NetR` subsystem. TimeSys Linux showed similar behavior as that of Linux/RK of Carnegie Mellon in UDP latency and sending-side throughput measurements.

We ran receiving-side throughput experiments for TimeSys Linux running on Host Modern. The results in the absence of any contention were again similar to that of Linux/RK. Next, we study the advantages obtained by TimeSys Linux's use of preemptive kernel threads for each IRQ handler. We do this by setting the priorities of the IRQs and the bottom-half thread to be *lower* than those of CPU-reserved processes.

4.4.1 Throughput with greedy contention

This experiment was performed under the same setup as in Figure 6. The results obtained in the absence of a receiver on Host Modern are displayed in Figure 11. It shows that the received bandwidth under the `NetR` thread (with no `netR` reservation) went down beyond 12Mbps, which is slightly higher than that of standard Linux. But with the `netR` reservation, it equaled the sending bandwidth all the way up to 23Mbps.

The sudden drop in throughput beyond 15Mbps in Linux/RK did not occur in the case of TimeSys Linux simply because IRQs run as separate threads and hence the scheduler does not charge their activities to some other threads or processes. The throughput of the reserved flow did drop beyond 23Mbps due to the failure of the network interrupt handler thread to cope up with the excessive influx of packets. However, the combined throughput of the reserved flow and the greedy flow became constant (not shown).

4.4.2 Throughput with greedy receiver

This experiment was repeated with a receiver running on Host Modern that captured packets from Host Rembrandt. As shown in Figure 12, the results obtained under a `netR` reservation remained almost unaffected; although the results under only a `NetR` thread went down further mainly due to buffer overflow at the backlog queue. This again happens since the `NetR` thread ran at a lower priority than the CPU reserved application and hence got less time to process packets than the network bottom half in Linux.

Therefore, we can conclude that a `netR` reservation provides guaranteed processing time for the received packets in the presence of a greedy network traffic up to a high volume of bandwidth. Also, the `NetR` subsystem of TimeSys Linux performs significantly better than that of Linux/RK at Carnegie Mellon in terms of the operating range of bandwidth. This happens for two reasons,viz., (1)separate threads for each `netR` reservation provides much better isolation, and (2)IRQs run as separate kernel threads, and avoid anomalies in the system imposed by the scheduler.

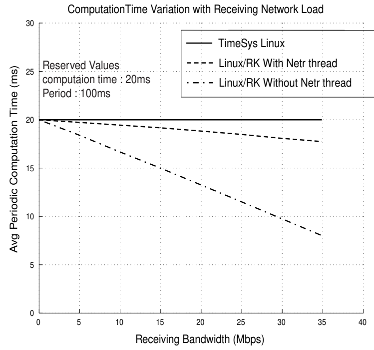

Figure 13: Computation Time Variation

4.5 Processor and Network Load

This experiment examines how a receiving network load affects a task under CPU reservation. We reverted back to the older experimental setup as shown in Figure 4.

Host Modern generated a network load on host Abel and Host Abel runs a periodic CPU-bound real-time task. The task was assigned a computation time (C) of 20 ms in a period (T) of 100 ms. In the standard Linux kernel, the processing times consumed by the interrupt handler and the bottom half were charged to the currently running thread, or in RK, the CPU reserve of the "current" *resource set*. We had inserted hooks at the device driver and the bottom half to account for their processing times and subtract them from the computation time of the resource set of the preempted process. Otherwise, RK would have incorrectly reported more CPU usage than the application consumed in reality. These hooks were *not* needed in TimeSys Linux as the interrupt handlers and the bottom half run as separate kernel threads and therefore do not cause any discrepancy in resource accounting.

The sending bandwidth from host Abel was again varied by means of the `network` reservation [8] scheme of RK. The results were taken under three system configurations on Host Modern, viz., Linux/RK and no `NetR` subsystem, Linux/RK and `NetR` subsystem, and TimeSys Linux. Figure 13 plots the periodic computation time (in ms) averaged over three hundred periods against the inbound network bandwidth (in Mbps). The plot clearly shows the degradation of computation time with increase in network load in Linux/RK of Carnegie Mellon. But it degraded at a much faster rate in the absence of the `NetR` subsystem. For example, at an inbound traffic of 35Mbps, the periodic CPU usage of the task dropped down to 17.8ms for RK with `NetR`, which is a 11% reduction with respect to the reserved value of 20ms. But the same with no `NetR` went down to as low as 7.8ms, a drop of 61% relative to the reserve. Finally under TimeSys Linux, the CPU usage remained *unaffected* at 20.0ms.

In the absence of the `NetR` subsystem, Linux/RK has no control on CPU-usage of the hardware and the software interrupt handlers. On the other hand, the `NetR` subsystem is able to control the packet processing time at the software interrupt level (bottom half) but fails to control the hardware network interrupt handler activity. But TimeSys Linux is able to control both the hardware and software interrupt activities. Setting their priorities lower than a CPU-reserved task, it is able to provide undiminished QoS to the CPU-reserved task.

5 Conclusion and Future Work

We investigated the problem of cooperative scheduling of processor cycles and the network bandwidth and introduced a new real-time network subsystem based on Linux/RK. We propose the use of kernel threads for network protocol processing and using processor reservations for controlling different flows. Our experiments show that these features result in guaranteed and protected processing of received network data. They also offer increased network throughput and stability under high load.

Our results demonstrate the necessity of control and precise accounting in the processing of interrupt handler. Because of this, the `NetR` subsystem of TimeSys Linux outperforms that of Linux/RK of Carnegie Mellon. The last experiment showed that a CPU-bound real-time task could potentially miss deadlines if the interrupts generated by the packets received from the network remains uncontrolled.

We also investigated the behavior of a typical network packet scheduler which controls the flow of outgoing packets in presence of a deluge of incoming packets. We found that the packet scheduler fails to perform correctly under this situation, even when the incoming packets and the outgoing packets do not share the *same* physical channel. In other words, in a PC with a typical multiport network adapter, the flow of incoming packets on one port of the adapter can potentially hinder flows of outgoing packets from another port. The solution lies in complete separation of threads of control on each port so that the interrupt handlers for each port can be scheduled differently.

Alternatively, one could consider using dedicated co-processors for individual network cards. While relatively more expensive, this is not a perfect solution either. A precise solution would require one network co-processor for each port of the multiport network cards. Thus, individual interrupts from each port can be completely isolated from the main processor as well as from each other.

Our future work will include the development of a more sophisticated network subsystem that will provide more control over sending and receiving of data across the network, namely guaranteed service to reserved network packets to be sent across the network in the presence of greedy receiving network load and subsequent extension of this scheme across PC-based routers and gateways.

References

[1] R. Rajkumar, K. Juvva, A. Molano and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-time and Multimedia Operating Systems. In *Proceedings of the SPIE Conference on Multimedia Computing and Networking*, January, 1998.

[2] S. Saewong and R. Rajkumar. Cooperative Scheduling of Multiple Resources. In *IEEE Real-Time Systems Symposium*, December 1999.

[3] S. Oikawa and R. Rajkumar. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. *In proceedings of the 1999 IEEE Real-Time Technology and Applications Symposium*, June 1997.

[4] C. Lee and K. Yoshida and C. Mercer and R. Rajkumar. Predictable Communication Protocol Processing in Real-Time Mach. *In the proceedings of IEEE Real-time Technology and Applications Symposium*, pp. 220-229, June 1996.

[5] P. Druschel and G. Banga Lazy Receiver Processing: A Network Subsystem Architecture for Server Systems. *Usenix Symposium on Operating System Design and Implementation*, pp. 261-275, October 1996.

[6] K. Jeffay, F. D. Smith, A. Moorthy and J. Anderson. Proportional Share Scheduling of Operating System Services for Real-Time Applications. *IEEE Real-Time Systems Symposium*, December 1998.

[7] J. Brustoloni, E. Gabber, A. Silberchatz, A. Singh. Signaled Receiver Processing *Proceedings of the USENIX 2000 Annual Technical Conference*, June 2000.

[8] S. Ghosh, R. Rajkumar. Network Bandwidth Reservation Using Rate-Monotonic Model. *IEEE SoftCOM*,pp. 363-370, October 1999.

[9] G. Banga, J.C. Mogul. Resource containers: A new facility for resource management in server systems. in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[10] J. Mogul and K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Transactions on Computer Systems*, Vol. 15, No. 3, pp. 217-252, August 1997.

[11] C. W. Mercer, S. Savage and H. Tokuda *Processor Capacity Reserves: Operating System Support for Multimedia Applications.* IEEE International Conference on Multimedia Computing and Systems, pp. 90-99, May 1994.

[12] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. *In the proceedings of Multimedia Japan 96*, April 1996.

[13] C. Maeda and B.N. Bershad. Protocol service decomposition for high-performance networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles.* pages 244-255, 1993.

[14] K.K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proc. Globecom'92 IEEE Global Telecommunications Conference*, pages 622-626, December 1992.

[15] Ttcp benchmark tool,at http://www.ccci.com/ product/network_mon/tnm31/ttcp.htm

[16] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, H. Bohme. *Linux Kernel Internals* Addison Wesley, 1998.