End-System Support for Idle-Time Networking

Lars Eggert and Joe Touch

USC Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292-6695, USA
{larse, touch}@isi.edu

May 14, 2001

Abstract — **Processing and transmission of idle-time network traffic ideally only occurs when resources would have been idle in its absence. Proposed extensions to the Internet service model are similar to idle-time networking, but focus on network support. This paper investigates end-system extensions needed under such a service model. It introduces a simple model with two preempting prioritized traffic classes (regular and idle-time). Experimental results show that current OS mechanisms cannot provide effective idle-time service. Analysis of OS network processing identifies its event-driven nature as the key issue. Experiments with a proof-of-concept implementation of minimal extensions for idle-time networking show them more than 97% effective in isolating higher-priority traffic from the presence of concurrent low-priority traffic.**

Index Terms — **idle-time, preemption, quality of service, precedence, differentiated services**

I. INTRODUCTION

Ideally, in a network with support for idle-time use, lower-priority packet processing will only occur when resources would have been idle in the absence of such traffic. Consequently, the presence of lower-priority traffic would be undetectable when observing higher-priority traffic transmissions. In such a network, lower-priority classes can only use resources not already consumed by higher priorities. Starvation may occur: If higher-priority traffic saturates a link, lower-priority traffic will not receive service.

With idle-time networking (ITN), packets of different priority classes experience different per-hop forwarding behaviors. Packets queued at a router are transmitted in order of decreasing priority, and lower-priority packets are dropped from a full queue when higher-priority packets arrive.

Some proposals to extend the Internet for differentiated services are similar to this architecture. However, most research in that area has focused on network support for these mechanisms. This paper instead focuses on end-system extensions required to provide end-to-end network service with support for idle-time use.

Experimental results presented in the next section show that current operating systems are not effective in establishing such different service levels for network traffic. The event-driven, asynchronous nature of network stack processing interferes with attempts to use CPU-scheduler-based mechanisms as offered by current systems to control network send behavior.

Observations gained during an analysis of network stack operation form the basis of a design to support idle-time networking (ITN), comprising of a minimal set of extensions to the current BSD network stack. These modifications concentrate on the sender's network layer; transport protocols and socket API remain unchanged.

Experimental results obtained from a prototype implementation of the new mechanisms in the BSD network stack suggest that they are effective in establishing idle-time network service: Using the new mechanisms, higher-priority senders can achieve 97-99% of the throughput in the basic case, effectively isolating them from the presence of concurrent lower-priority traffic.

II. FAILURE OF EXISTING SCHEDULERS

One of the main tasks of an operating system (OS) is to control and schedule application access to host resources. To support a wide variety of applications, a general-purpose OS employs simple and predictable schedulers, trying to provide fair service to all users of a resource.

Since the CPU has traditionally been the bottleneck resource in a system, its scheduler is more evolved than those for other resources are. UNIX systems use a multilevel feedback queue [3], which favors interactive, bursty processes (which do not fully utilize their allocated CPU quantum) over compute-bound batch jobs (which do). It rewards bursty processes by increasing their priority, and punishes compute-bound ones by lowering it. Most I/O-bound processes are bursty – they block during device operations – and thus achieve high CPU priorities.

Commonly, the CPU scheduler offers the user processes

some degree of control over their priorities through the *nice* utility. Non-privileged processes may thus lower their priority from the default (increasing the priority is restricted to privileged processes).

Some POSIX-compliant systems [2] offer three distinct priority classes for processes (real-time, regular and idle-time), each managed by its own multilevel-feedback queue. Processes in higher classes preempt any lower-class ones; starvation of lower-class processes occurs when high-class load increases to saturation.

Simple first-in-first-out (FIFO) schedulers organize access to most other resources. While FIFOs by themselves do not assure fairness, they can do so in combination with a fairness-enforcing CPU scheduler (since a process cannot issue any resource requests without a CPU to run on). These other resource schedulers typically do not allow processes to influence their scheduling decisions. Thus, current systems offer only two candidate mechanisms (*nice* and POSIX scheduling) to implement ITN.

After defining the ITN model in more detail in the next section, experiments with the existing two CPU-scheduler-based mechanisms show that neither is sufficient to prioritize network traffic into two service classes for effective idle-time use.

A. Idle-Time Network Model

The ITN model used throughout this paper is a simple extension of the current Internet service model, where routers (and hosts) treat packets equally according to a *best-effort* discipline [4]. Note that ITN does not change this fundamental model: The network may still reorder, drop or duplicate packets. Idle-time networking is strictly a per-hop function of giving higher processing preference to certain packets.

In the idle-time network model, packets belong to either of two classes: foreground (FG) or background (BG). Ideally, BG packet processing will only occur when resources would have been idle in the absence of BG traffic. Thus, the presence of BG traffic would be undetectable when observing FG traffic transmissions. Under real conditions (non-interruptible packet transmissions, non-zero-cost queue operations), complete isolation of FG traffic is difficult to achieve.

Router support for ITN is simple: A Router will always forward all FG packets in its queue before any BG packet, and it will drop BG packets from a full queue to make room for arriving FG ones. In other words, ITN replaces a router's FIFO queue with a two-layer priority queue. FG packets continue to experience *best-effort* service, while BG packets see sub-*best-effort* (i.e. *least-effort*) service. This is not a new idea: The original IP specification [8] contains support for a *precedence* field in the datagram header to indicate dropping and forwarding priorities.

More recently, some of the proposed extensions to extend the Internet to support differentiated services [11] are similar to the idea of ITN: *Expedited forwarding* (EF) [1] redefines a value in the IP *type-of-service* field to mark some packets with a higher forwarding priority. It also suggests configuring a rate limit for expedited packets, in order to prevent starvation of lower-priority traffic. While EF focuses on providing virtual leased lines with a fraction of the capacity of the physical link, in the absence of a configured rate limit for expedited traffic it becomes one possible implementation of ITN: Expedited packets belong to the FG class, and regular packets belong to the BG class.

Idle-time networking can also be seen as a combination of two other proposals from the differentiated services community: One is marking packets as *in* or *out* at routers [9], indicating whether they are in compliance with their assigned traffic class. During congestion, packets marked as *out* are give drop preference (similar to ATM's *cell-loss-priority* bit [12] or frame relay's *discard-eligible* bit [13]). The other proposal is a scheme where routers forward packets in strict order of priority [10]. Together, these proposals can implement ITN by giving drop preference and lower forwarding priority to BG packets.

In a previous paper, we have investigated the idea of idle-time network service at the application layer, by distinguishing between FG and BG web transactions [20]. The LSAM project [24] built on this idea and used BG multicasting of web transactions to pre-load self-organizing, distributed caches with popular content.

B. End-System Support

The network stack of an end-system sending or receiving prioritized traffic must implement the same outbound and inbound processing mechanisms as routers in the same service model. However, while routers only need to concern themselves with prioritizing packets during forwarding, the situation on end-systems is more complex: Routers operate at the network layer, while packet processing on end-systems covers the whole range of the protocol stack. Thus, end-systems need to satisfy additional requirements to support end-to-end ITN.

To generate packets, processes need CPU time and possibly other resources (likewise for receiving packets.) Thus, simply replacing the FIFO of a network interface with a priority queue – which enables ITN on routers – is not enough. Other *backgrounding mechanisms* are required to guarantee that BG traffic does not cause drops or delays for FG packets.

Two simple backgrounding mechanisms available on current systems include running the BG sending process at *nice* or POSIX CPU priorities. The following section presents experimental results showing that both these mechanisms are ineffective in establishing idle-time network service for the BG class; Section III analyzes the reasons in more detail.

C. Experimental Setup

In these experiments, two copies of the same benchmark process run in parallel on a single host. The process is network-bound; it simply tries to send as much pre-generated random data to a second machine as possible. At the end of the experiment, the process reports the amount of data successfully sent. One of the two benchmark processes is the FG sender, the other one the BG sender.

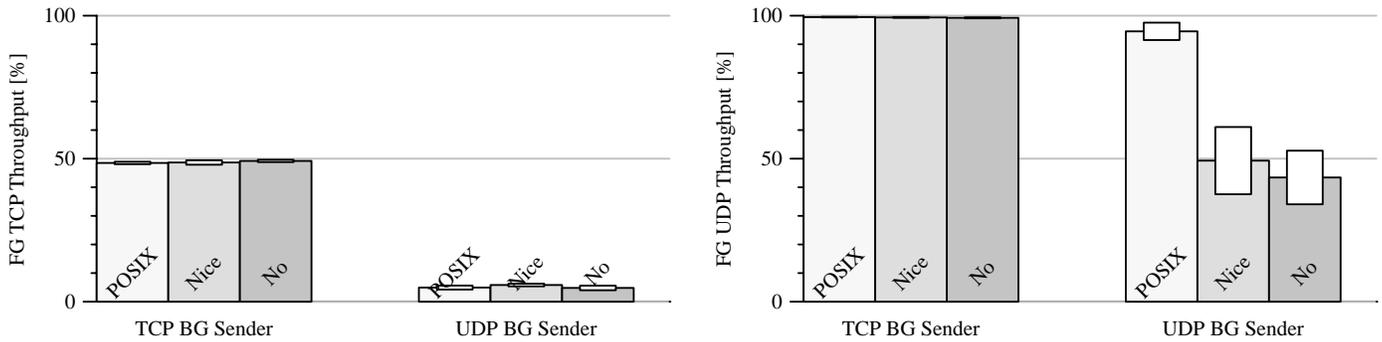Each benchmark process uses a fixed number (here: 3) of

Figure II.1 Normalized mean throughput of a FG sender under unlimited load in the basic case ("No") and with two backgrounding mechanisms ("Nice" and "POSIX"), using TCP (left graph) and UDP (right graph) with 95% confidence intervals.
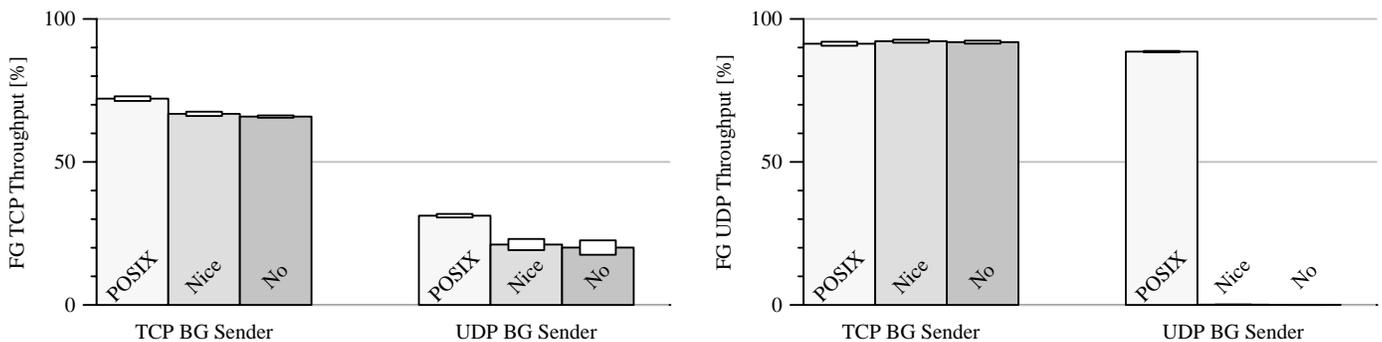

Figure II.2 Normalized mean throughput of a bursty FG sender in the basic case ("No") and with two backgrounding mechanisms ("Nice" and "POSIX"), using TCP (left graph) and UDP (right graph) with 95% confidence intervals.

either TCP or UDP connections to send its traffic, since a single TCP connection cannot easily overload an isolated network link due to TCP's congestion control algorithm. When sending with TCP, the benchmark blocks until one or more connections become writeable, writes on those descriptors and starts over. When using UDP, it sends one message over each descriptor until the send call fails with an indication that the outbound device queue is full. It then sleeps for 10ms, and starts over[2].

Another variable is the *intensity* of the FG sender[3], which controls how large a fraction of its time quantum a benchmark process spends in the previously described sending loops. For a fraction of 0.1, for example, the process will only try to send traffic for 10% of its allocated time quantum. On BSD systems, the default quantum is 100ms, meaning the benchmark will generate send bursts of 10ms before sleeping for 90ms.

For each combination of transport protocol (TCP and UDP) and send intensity (full: intensity = 1 and light: intensity = 0.1), the experiment is run for 1 minute. Throughput numbers are normalized against the maximum throughput a lone FG sender achieves with no BG traffic present. The graphs in the remainder of this paper show mean normalized throughputs

with 95% confidence intervals over a series of 10 iterations.

The sending host (running the two load-generators) and receiving host are two identical FreeBSD 4.2-RELEASE machines with 300Mhz Pentium II processors. They are located on an isolated, switched, full-duplex 100Mbps Ethernet. This setup is network-bound; one machine can satiate the link with a CPU load of 55%.

As a metric for the effectiveness of ITN support, we compare the throughput of the FG sender in the presence of a BG sender using one of the investigated backgrounding schemes against the basic case (no BG sender present). Better ITN mechanisms will yield higher FG throughputs. With optimal ITN support, the FG sender should reach 100% of the throughput it achieves with no BG traffic present.

D. Evaluation
1) Unlimited FG Load

In the first experiment, the FG sender sends TCP traffic at full intensity to the receiver. The left diagram in Figure II.1 shows the measured and normalized throughput rates together with 95% confidence intervals (narrow white bars overlaying the wider gray bars).

With a BG TCP or UDP sender, neither the POSIX nor the *nice* backgrounding mechanism can establish idle-time network service that significantly improves on the basic case (the differences in throughput lie within the confidence intervals of the measured series). Furthermore, for a UDP BG sender, this experiment demonstrates the worst-case scenario:

---

[2] This emulates the sending behavior of the ping utility in "flood" mode, and generates enough traffic to satiate the 100Mbps link: 50 packets (length of device queue) * 1500 bytes (Ethernet MTU) / 10ms = 600Mbps send rate.

[3] The BG sender always sends at full intensity to simulate the worst-case situation for FG senders.

a BG sender without rate-control can virtually shut down FG service. An effective ITN mechanism must adapt to this scenario. Both examined schedulers fail to do so; the FG sender only achieves around 5% of the possible throughput.

The right diagram in Figure II.1 shows the case of a FG UDP sender under BG traffic created by TCP or UDP sources. With a TCP BG sender, this case is the inverse of the worst-case scenario presented above: Here, the FG UDP sender monopolizes the link: FG throughput is over 99% for all three cases, even the basic one.

If the BG sender also uses UDP, the POSIX scheduler noticeably outperforms the *nice* one (90% throughput versus 50%). Variations in throughput are also higher, as indicated by larger confidence intervals.

2) Bursty FG Load

In the second experiment, the FG sender is only active for 10% of its time quantum (= 10ms). Figure II.2 shows the results for this case. When the FG sender uses TCP to transmit its traffic bursts, the POSIX backgrounder offers small FG performance improvements (5-10%) over the basic case for both TCP and UDP BG senders. Throughput does not increase with the *nice* backgrounding mechanism. With a UDP BG sender, the POSIX backgrounder increases performance 90% over the basic case and *nice* mechanism. The latter are extremely ineffective in giving higher priority to bursty FG traffic; it only achieves 1-3% throughput.

E. Discussion

The experimental results presented in this section demonstrate that current OS mechanisms are not sufficient to implement ITN service. While scheduling BG senders at POSIX idle-time priority improves FG performance in some scenarios (mainly for UDP senders), even then total FG throughput only reaches 90% of the maximum. For other cases (FG TCP senders), the POSIX backgrounder only offers small performance improvement of 5-10%, increasing total throughput to 5-70%.

An effective mechanism to support ITN would achieve FG send performances close to 100% in all of the above scenarios. The next section will examine the details of the BSD network stack and discuss why current mechanisms fail to provide functional support for ITN.

III. CURRENT NETWORK PROCESSING

The experiments in the previous section have shown that current OS mechanisms (*nice* and POSIX scheduling) are not sufficient to realize ITN. This section will analyze the reasons of this failure by tracing the path of outgoing and incoming data through the BSD network stack, and pinpoint issues that inhibit ITN. Figures III.1 and III.2 give a (simplified) view of the flow of execution inside the network stack during outbound and inbound processing, while Figure III.3 shows the data flow through its various buffers. This analysis forms the basis of the OS modifications discussed in the next section.

A. Outbound Network Processing

All user-level socket output flows through the `sosend()`
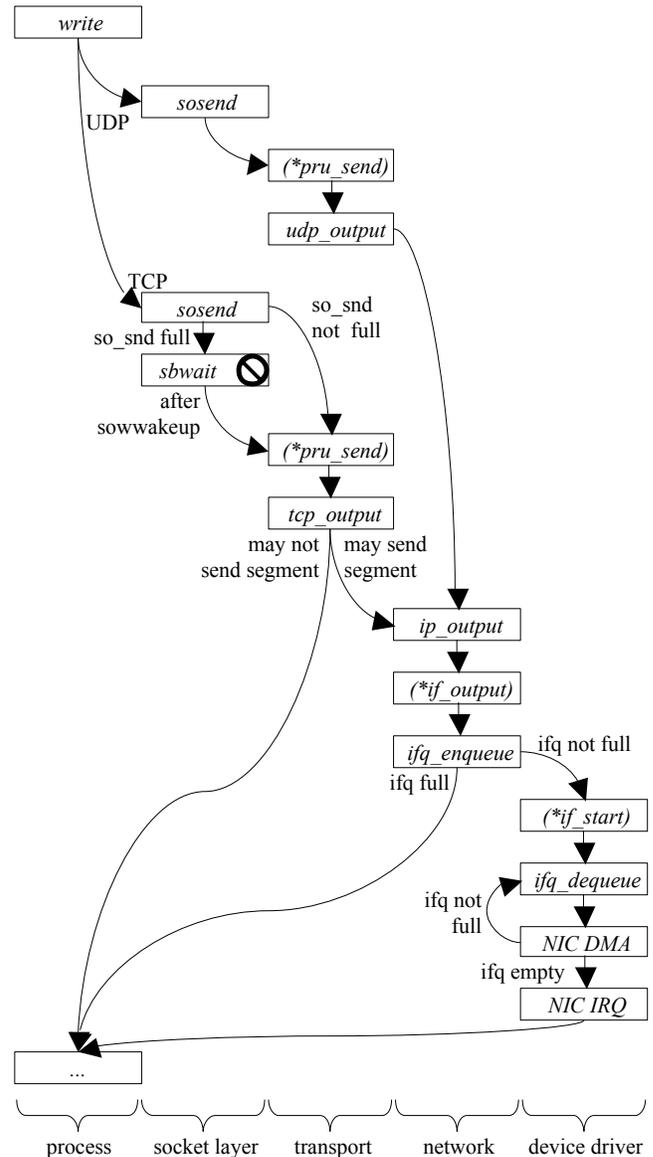

Figure III.1 Network stack outbound data processing.

function in the kernel down into the kernel (see Figure III.1). Depending socket's protocol and domain, it then calls the appropriate transport-layer output function through a dispatch table. For the Internet protocols, those are `udp_output()` and `tcp_output()`.

TCP sockets must maintain a copy of the user data so TCP's recovery algorithm can retransmit the contents of lost packets. Every socket contains a send buffer (`so_snd`) for that purpose. If the send buffer is full, `sosend()` will block the sending process until the buffer drains. When the send buffer has enough space available, `sosend()` appends a copy of the user data to it, and then calls the transport-layer output function `tcp_output()`. Inside `tcp_output()`, the protocol checks if it may send a segment for the respective connection, according to its congestion control algorithm and timeout rules. If so, `tcp_output()` calls the network-layer output

function `ip_output()`; if not, the system call is complete and process execution continues after the write system call.

When a process writes on a UDP socket descriptor, `sosend()` does not buffer any data. UDP as a simple, unreliable datagram protocol does not offer protection from packet losses. Instead, `sosend()` immediately calls the transport-layer function `udp_output()`, which in turn simply calls the network-layer output function, `ip_output()`.

Network layer processing for both UDP and TCP is identical. At first, `ip_output()` performs a route lookup, and then tries to enqueue the data in the device queue of the outgoing interface for that route[4]. If the device queue is full, `ip_output()` drops the packet and the write system call is complete[5].

If `ip_output()` was successful in enqueueing the packet, it calls the output function of the outgoing network interface (`*if_output`). This function, in turn, checks if the hardware is ready to transmit data, and if so, dequeues a packet from the device queue and starts transmission (`*if_start`). If not, it will simply return.

After transmission starts, the driver will repeatedly dequeue and transmit packets until the device queue is empty (or the hardware's send buffer is full). It is important to note that the driver code runs at one of the highest interrupt priority levels (most interrupts are blocked), and so usually cannot be interrupted until the device queue is drained completely.

B. Inbound Network Processing

Inbound network stack processing starts with the physical reception of a packet by the network device (see Figure III.2). The device will signal the availability of data to the kernel by issuing a device interrupt, which is handled by the device driver's interrupt routine. It copies the data from the device memory into main memory. The input routine of the driver then enqueues the data into the correct protocol receive queue. All IP data demultiplexes into the incoming IP queue (`ipintrq`) and a software interrupt signals data arrival to the upper half of the kernel. If `ipintrq` is full, the driver drops the data.

At this point, processing loops back to the driver's interrupt handler. While more packets are ready to be transferred from the device memory, the driver will continue to demultiplex and enqueue them for reception by higher-level protocols. Again, since the driver runs at one of the device interrupt priority level (IPL), it will not exit this loop until the device receive buffer is empty[6].

When a software interrupt signaling IP packet reception occurs, the `ipintr()` handler loops over all packets in the IP
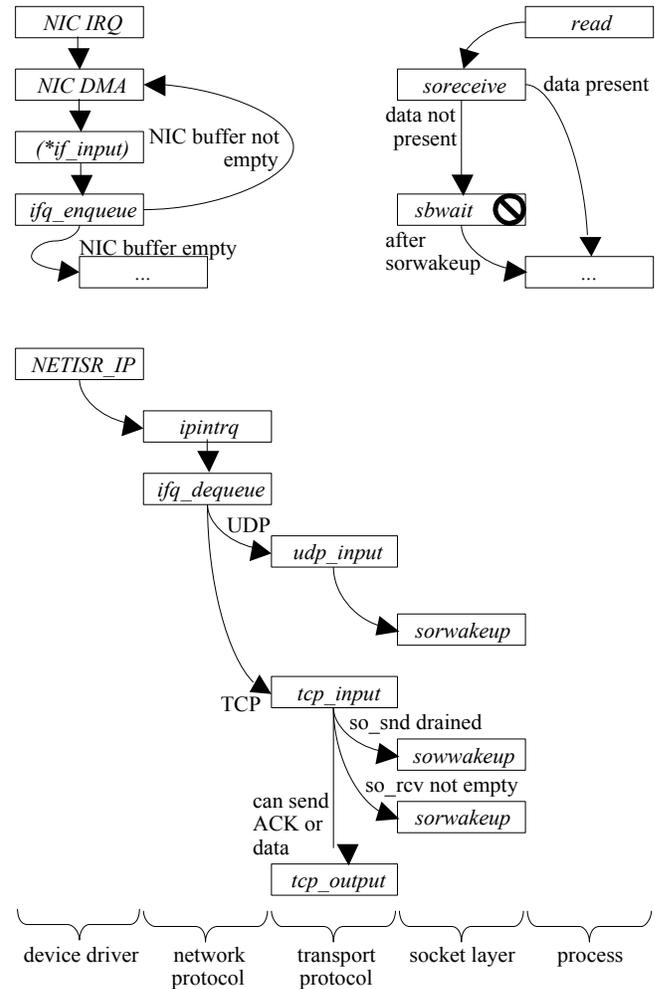

Figure III.2 Network stack inbound data processing.

incoming queue and calls `ip_input()` for each one. That function discards corrupted packets, dispatches packet forwarding (if needed) and manages fragment reassembly. For a packet destined for the local host, it calls the transport-layer input routine, based on the packet's protocol field.

If dequeueing a UDP packet, `ip_input()` dispatches the packet to `udp_input()`, which appends the data to the receive buffer of the corresponding socket, and unblocks processes blocked to read data (`sorwakeup`).

When `ip_input()` dequeues a TCP packet from `ipintrq`, it passes it to `tcp_input()`. As part of TCP protocol processing, `tcp_input()` may trigger sending new TCP packets (data and/or ACK) by calling `tcp_output()`, and wake up processes waiting to enqueue more data into the send buffer (`sowwakeup`). Data flows out of `tcp_input()` along the same path it does for UDP: the routine copies it into the receiving socket buffer, and waiting processes are unblocked (`sorwakeup`).

Whenever a process reads from a socket, `soreceive()` checks if enough data is present in the socket receive buffer to satisfy the read request. If so, it copies it to the process buffer

---

[4] Part of this processing happens outside the actual driver by helper functions that implement the hardware-independent processing for a family of similar interfaces (e.g. ether_output() for all Ethernet devices).

[5] This is how UDP packets are lost at the sending host without ever entering the network.

[6] Under high enough loads, this can lead to receive livelock [5], when the OS spends all processing capacity on receiving and dropping packets (since the higher-level incoming queues eventually fill up and no processes can be scheduled to drain them).
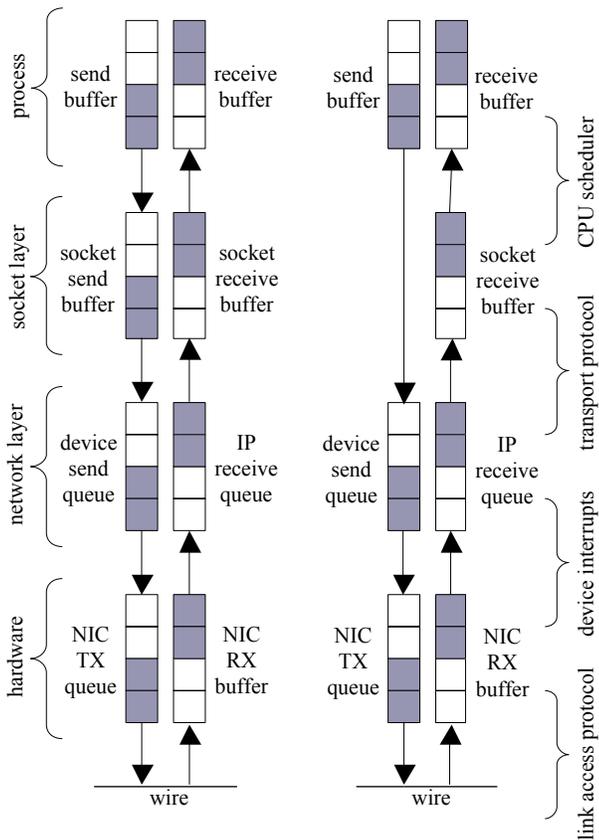
Figure III.3 Queueing at different layers for TCP (left) and UDP (right) processing.

and returns. If not, it blocks execution until the transport layer signals the arrival of more data through `sorwakeup`.

C. Discussion

Processing in the BSD network stack (and similar UNIX-like operating systems) is an intricate combination of queueing, timeouts, interrupts, and blocking and resuming processes. Synchronous events (CPU scheduler) as well as asynchronous ones (timeouts and device interrupts) cause data to flow between the various buffers in the system.

1) UDP Processing

As described in the previous section, UDP data written by a process will usually go directly into the outbound device queue. It may seem that if the CPU scheduler enforced strict priorities, UDP data send by a lower-priority process could never interfere with that of a higher-priority one, because the priority CPU scheduler would never allow the lower-priority sender to execute. However, the POSIX scheduler results in Section II.D demonstrate that this is not the case.

The reason lies in the way typical UDP senders are implemented: In essence, UDP senders limit their send rate by blocking for a period of time when the send system call indicates a full device queue. (If this never happens, the outgoing link speed is higher than the data rate of the sender.) If the device queue fills up before the time quantum of a process runs out, it will sleep, causing the CPU to context-switch to another process. Even under the POSIX scheduler, if a higher-priority process voluntarily sleeps, lower-priority ones may run.

As noted above, the lower half of the kernel runs at IPL asynchronously from scheduled events in its upper half. This means that when the new process starts its time quantum, the driver has usually drained some packets from the device send queue and more data can be enqueued. So a BG sender scheduled after a FG sender sleeps because of a full device queue can usually send at least some packets before the queue fills up again, and it in turn sleeps.

2) TCP Processing

For a TCP sender, the kernel buffers data in the socket send buffer, which the transport layer drains according to TCP's congestion control and timeout rules. The write call succeeds after the data enters the socket buffer, and the process continues execution.

Either timeouts (in-kernel timer firing) or ACK receptions (device interrupt) trigger TCP packet sends. Both of these events happen independently from CPU scheduling. The handlers for both events run at higher IPLs than user-level processes, and will thus interrupt process execution. This means that a process may not even be running when the kernel sends packets on its behalf.

An ITN mechanism based on a modified CPU scheduler (like *nice* and POSIX) cannot hope to regulate network transmissions in this feedback system. It only controls which candidate process can access the socket queues to enqueue or dequeue data, not the timing of the transmission of that data.

IV. OS EXTENSIONS FOR IDLE-TIME NETWORKING

The key issue with the two CPU-scheduler-based candidate mechanisms to implement ITN is the event-driven nature of kernel network processing. Nearly all network routines – with the notable exception of UDP sends – happen asynchronously with user mode execution: device interrupts trigger packet transmissions and receptions. Packet receptions trigger incoming transport protocol processing, which in turn may unblock processes waiting for data reception on a socket. For TCP, packet receptions (and to a lesser degree, kernel timeouts) trigger packet sends.

In a sense, the network stack is an event-based system, where event priorities are equivalents to the IPL of the corresponding handlers. As demonstrated by the experimental results in Section II.D, the previously examined CPU-scheduler backgrounding mechanisms have only very limited impact in such a system.

A second issue is the use of FIFOs for all kernel queues. The processing order of a FIFO queue is identical to the enqueue order, which may cause a queue's consumers to process earlier arriving BG data before FG (e.g. a FIFO device queue may send BG data before FG data, because it was enqueued earlier). This must not occur in a system supporting ITN.

A. Design Goals

The network stack is a complicated system, and many

applications rely on its API (socket interface) and service semantics. Therefore, it is critical to avoid fundamental changes to the network stack. Additionally, much effort went into designing and fine-tuning the Internet's transport protocols. OS extensions for ITN must not modify these transport protocols, to avoid incompatibilities with current standards.

It is also impractical to change all network drivers to support ITN, so hardware-dependent driver code must not change for ITN extensions. Note that part of the driver code is common to all devices of the same family; these routines could be safe to modify.

In addition, for end-to-end ITN, routers in the network must distinguish between FG and BG packets, as described in Section II.A. The focus of this paper is in host extensions, so it assumes network support for ITN is available and the network handles packets according to their service marks.

In summary, a design for OS extensions for ITN must be a simple extension of the current socket layer, must not modify the transport layer, and must not require changes to the hardware-dependent parts of device drivers – consequently, they must mainly extend the network layer.

B. Design

1) Queueing Support

One issue identified earlier in this section was the use of FIFOs for all queues in the network stack. To support ITN, two-level priority queues must replace most FIFOs in the network stack (see Section II.A).

Part of the KAME IPv6/IPsec package [7] for BSD is the ALTQ framework [6] of alternate queueing disciplines. ALTQ replaces the outgoing standard FIFO queues of device drivers with configurable queueing disciplines, including priority queues. We have extended ALTQ to the inbound protocol queues (mostly `ipintrq`) and to drop lower-priority packets for higher-priority ones when the queue is full. ALTQ filters put marked packets into a lower-priority traffic class, for both outgoing device and incoming protocol queues.

2) Socket Layer Support

The service level of the network stack must not decrease for ITN-unaware applications – the kernel must not send their packets as BG by default. Only ITN-aware applications may use the new service class, by explicitly indicating this to the kernel. The socket layer offers socket options to set user-configurable options on a per-descriptor basis. Thus, the only socket-layer change needed is a new socket option (SO_BACKGROUND) that indicates that the network stack should treat all traffic from or to a socket as low-priority BG traffic.

Note that this scheme is the inverse of other proposals for packet marking that use marks to increase the service level (e.g., expedited forwarding). Without proper policing mechanisms, these schemes become problematic – nothing keeps processes from marking all their packets as high-priority, and thus receive better than best-effort service. The proposed marking scheme for ITN avoids these complications by only allowing applications to lower their service level[7] (to least-effort BG service.

3) Network Layer Support

Because of the event-based nature of the lower half of the kernel, drivers will transmit packets as soon as they enter their device queue (a transmitter activation follows each enqueue operation). Since the driver code executes at a higher IPL than the network layer, it typically sends the packet before another one can be enqueued[8]. Consequently, the network layer must verify if BG packets may be send at a particular time (see below) before it enqueues them into the device queue.

The key idea is that the host should never send BG packets to any destinations when a FG sender is using the same outgoing interface. Instead, the network layer should drop these BG packets, signaling an *out of buffers* (ENOBUFS) error condition. UDP senders must already be prepared to handle this error condition (it occurs when the device queue fills up), and TCP will take the packet drop as an indication of congestion and lower the rate of the BG sender.

There are several possible methods to determine if an interface is in use by a FG process before enqueueing a BG packet into a device queue. The simplest one is to check if a FG protocol control block (PCB) exists that uses the same outgoing interface[9]. While this simple approach is effective, it is also too restrictive: A single FG TCP connection prohibits any BG traffic from being sent – even when it is idle.

A more effective identifier of active senders would not only check for the presence of a PCB for an outgoing interface, but also use additional means to determine if the PCB is an active user of the interface. For example, it could check if the corresponding socket had any queued data in its send buffer, which would indicate an active sender. The prototype implementation evaluated in the next section uses this technique.

Active UDP senders are more difficult to identify. Unlike TCP, UDP does not buffer any data at the socket layer (all UDP socket send buffers are always empty), so the check described for TCP in the previous paragraph is not effective. Furthermore, UDP writes are non-blocking; they either succeed in enqueueing data into the device queue or fail and return to the user process with an error code. No kernel state exists that allows determining precisely if a UDP sender is active or not at any given time.

The current design for ITN thus uses the following heuristic to check for active UDP senders: For each UDP PCB, the network layer will check if the corresponding process is sleeping or not. A sleeping process indicates (paradoxically) an active UDP sender. This heuristic depends on the common structure of implementing UDP clients, which send until they fill the device queue or run out of data, then sleep to enforce a send rate limit. (See the next section.)

---

[7] Similarly to the UNIX *nice* mechanism for changing CPU priorities, that allows a user process to only lower its processing priority.

[8] The queue only builds up when the link is busy (i.e. the transmitter cannot start) or the device send buffers are full.

[9] Its corresponding socket not having the SO_BACKGROUND socket option set identifies a FG PCB.
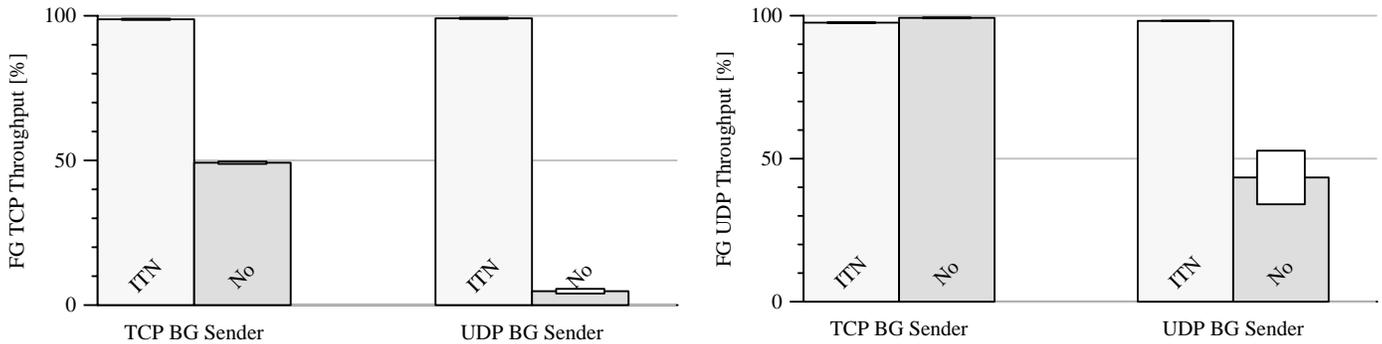
Figure V.1 Normalized mean throughput of a FG sender under unlimited load in the basic case (*No*) and with ITN backgrounding mechanism (*ITN*), using TCP (left graph) and UDP (right graph) with 95% confidence intervals.
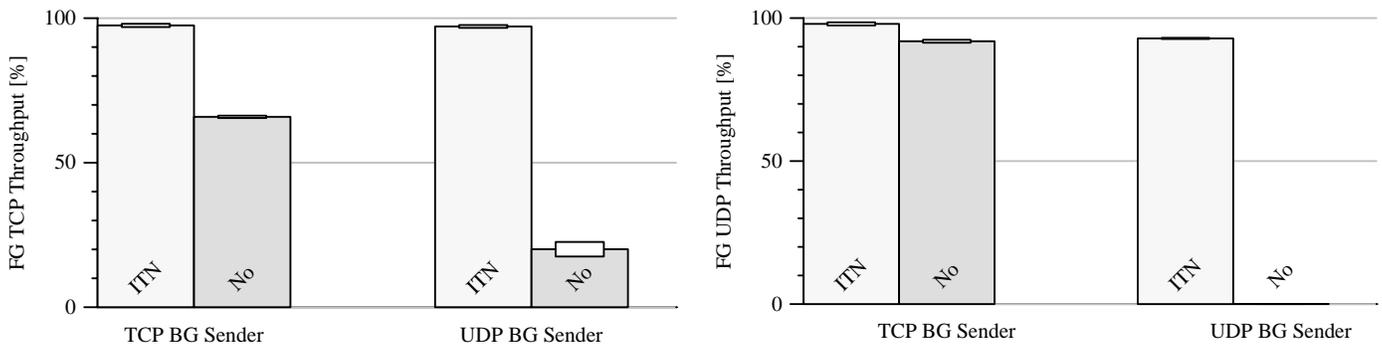

Figure V.2 Normalized mean throughput of a bursty FG sender in the basic case (*No*) and with the ITN backgrounding mechanism (*ITN*), using TCP (left graph) and UDP (right graph) with 95% confidence intervals.
.

C. Discussion

The design for ITN in this section is clearly a proof-of-concept, and practical reasons argued for minimal modifications to the current network stack. A completely redesigned network stack with support for ITN from the ground up would be an equally possible solution, but the extent of such an effort is outside the scope of this project.

Several performance issues exist with the current design. One is that the decision to enforce ITN at the network level causes BG packets to go through socket and transport layer processing, only to be dropped when FG senders use the same outgoing interface. Enforcing ITN at a higher layer would not incur this performance hit. For more compute-intensive future transport protocols (e.g. encrypted or tunneled flows), this may prove problematic.

A second performance issue is the per-BG-packet overhead of looking up the PCB for a packet and determining if FG senders (PCBs) exist for the same interface. The current implementation adds list of users (pointers to PCBs) to each interface to limit the impact of this search. Schemes that are more complex may further mitigate this overhead, but are outside the scope of the initial implementation.

Detecting active UDP senders (to protect FG UDP traffic from BG interference) at the network layer is difficult, due to lack of information. The kernel can gain information about TCP connections and their corresponding processes from internal state. For UDP senders, no such state exists at the

kernel level; UDP senders manage it inside the application. One future possibility could be to extend UDP to utilize to queue data at socket send buffer, and to drain it as the interface queue empties. This would allow the TCP technique to check for active senders to extend to UDP senders.

V. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of the ITN mechanism designed in the previous section, we repeat earlier experiments (see Section II.C) with the new ITN backgrounding technique. The experimental setup is unchanged, except that the BG senders are now using the new network ITN backgrounding method.

A. Unlimited FG Load

The left graph in Figure V.1 shows how a fully loaded TCP FG sender behaves under BG load generated by TCP or UDP senders that use the ITN backgrounder. In both cases, FG throughput is over 99% of the maximum – a 50% improvement from the basic case.

The right graph of the same figure displays the result for a UDP FG sender. Again, FG throughput under full load reaches 97-99% for both UDP and TCP BG traffic. With TCP BG traffic, this is no improvement over the basic case, which is the worst-case scenario for TCP (see Section II.D). (In fact, throughput is 1-2% lower, maybe due to processing overhead of the *ITN* mechanism.) With a UDP BG sender, performance

increases by about 55% to 99%.

The idle-time network backgrounder is effective in isolating the FG traffic from the presence of any BG traffic in these four full-load scenarios.

B. Bursty FG Load

The next experiments look at the performance of a bursty FG sender using the ITN backgrounder. For a TCP FG sender (left graph in Figure V.2), the new mechanism improves FG throughput between 35-80% to 99% total for both TCP and UDP BG traffic.

For a UDP FG sender (right graph in Figure V.2), the ITN backgrounding method also increases throughput to 99% for both TCP and UDB BG traffic. With a TCP BG sender, this is a minor improvement of 5% over the basic case (again, this is the worst-case scenario for TCP). For a BG UDP sender, the performance increase is around 90% – bursty FG UDP traffic was almost denied service in the basic case, now its performance is close to optimal.

C. Discussion

The experimental results presented in this section show that the proposed ITN extensions are effective in isolating FG traffic from the presence of BG traffic. In all the investigated scenarios, FG performance reaches 97-99% of the basic case (where no BG traffic is present), effectively isolating FG packets from the presence of BG traffic.

While the benchmark framework used for these experiments is flexible, the current load-generating processes are very simple. This was a deliberate choice, to factor out secondary system interactions from the results. Future experiments should investigate the behavior of the ITN backgrounder under real workloads, such as supporting a web server with FG and BG service classes.

VI. RELATED WORK

A. Application Layer Mechanisms

Migrating sockets [16] and Real-Time Mach push most protocol processing out of the kernel and into user-level process. A rate-controlling network scheduler then controls send operations from multiple sources to meet pre-defined quality-of-services parameters. In a sense, this design is the inverse of ours: The design for the ITN extensions minimizes changes to the network stack, and targets hidden scheduling (processing due to asynchronous events) at the lowest possible layer. Migrating sockets, on the other hand, completely re-design the network stack, and minimize hidden scheduling by doing protocol processing at the user level, where CPU scheduling controls it.

Building support for different service levels into the application through rate or resource limits is another effective approach to establish network service at multiple levels [20][21], especially for dedicated systems (e.g., web servers). However, these mechanisms can only hope to control traffic at a relatively coarse granularity – once data enters the kernel, it will be sent, and can possibly interfere with higher-priority data.

B. Middleware Mechanisms

Middleware approaches [18] for quality-of-service support, interposed between the application and the kernel, promise effective support for different service profiles without kernel changes, and only minor changes to the applications. However, by layering them above the kernel, scheduling traffic on a per-packet scale – like ITN does – become unattainable. Instead, middleware approaches try to control traffic at a coarser granularity (e.g., flow-based), similarly to application-layer approaches.

C. Kernel-Based Mechanisms

Many proposals exist to augment current kernels for networking at different service levels. Soft-real-time kernel extensions propose allowing processes to control scheduling and resource allocation. AQUA [17] is a kernel-level framework that allows cooperating processes to dynamically negotiate their CPU and network I/O requirements with the kernel. If a resource becomes congested, AQUA notifies processes so they can adapt to the new service environment. OMEGA [15] is an end-system kernel framework supporting soft-real-time scheduling of CPU, memory and network resources. Its focus is on providing end-to-end quality-of-service for multimedia applications. OMEGA is similar to AQUA; applications dynamically negotiate their resource requirements with a quality-of-service-broker. This is a key difference to ITN, which does not require extensive application changes. In addition, processes need not inform the ITN mechanism of their resource requirements.

Waldspurger and Weihl present a proportional-share scheduler, and have applied it to control network transmissions in the Linux kernel [14]. Experiments show that they are successful in allocating different shares of the managed resource to different applications. Eclipse [22] augments a BSD-based OS with proportional-share schedulers for CPU, disk and network schedulers. The key difference between proportional-share schedulers and ITN is that the schedulers prevent starvation, which is essentially prohibits ITN. Proportional-share schedulers also depend on a policy that governs assignment of priorities to resource requests, which ITN does not require.

As mentioned above, some of the proposed Internet extensions for differentiated services [11] are similar to the idea of ITN, and could provide its required network support [1]. Few of the differentiated services proposals discuss end-system requirements for an effective end-to-end implementation of the new service model. ITN is one possible implementation to support for some of these proposals in an end-system.

VII. FUTURE WORK

As mentioned above, the benchmark processes used during the experiments throughout this paper are deliberately simple. The next set of experiments should investigate the behavior of the current ITN design under a realistic workload, such as supporting a web server with FG and BG service classes.

An ideal ITN system would only send BG traffic during

times when the network interface would otherwise have been idle. Comparing the packet-scale send behavior of an ITN-enabled host with BG traffic against a basic one without (under the same workload) would illustrate how close the current design approaches the ideal.

Another set of experiments should evaluate the effectiveness of the proposed ITN design for CPU- or disk-bound applications. The scheduling discipline of the bottleneck resource likely governs overall system behavior. For example, in a disk-bound process mix, the disk I/O scheduler will essentially control process scheduling – servicing one processes requests over that of another allow the former to progress, while the latter stalls. Generalizing the idea of ITN to idle-time use of other resources may improve performance in those situations.

Lazy receiver processing [23] demultiplexes the incoming packet stream at the link layer into channels according to their destination socket, and does receiver protocol processing at application priority. The main goal of this proposal is to increase system fairness and stability under increasing traffic load by shortening the time spent in response to device interrupts. However, shorter processing at high IPL may also increase performance of for ITN.

VIII. CONCLUSION

This paper presented a design for a network scheduler to support idle-time networking (where lower-priority traffic is only sent using otherwise idle resources), after experimental results showed that current OS mechanisms are unable to effectively support such a service model.

An analysis of the current BSD networking stack identified the asynchronous, event-driven nature of processing in the kernel to counteract most effects the current CPU-scheduler-based mechanisms. The resulting design of extensions to support idle-time network use comprises of a minimal set of changes to the current BSD network stack. The new mechanism leaves socket API and transport protocols unchanged – most changes concentrate on outbound processing at the network layer. At the core of the ITN extensions lies a mechanism to identify active users of a network interface, and only pass BG data down to the driver level when no FG senders are active for an outbound interface.

Experimental results with the new ITN method show it to achieve 97-99% of the maximum throughput of the basic case in all investigated scenarios, effectively isolating FG packets from the presence of concurrent BG traffic.

ACKNOWLEDGEMENTS

Kernel patches against the KAME networking stack of the prototype implementation and the benchmark software used during the experiments in Sections II.D and V are available from the authors' web site at http://www.isi.edu/larse/.

REFERENCES

[1] V. Jacobson, K. Nichols and K. Poduri. An Expedited Forwarding PHB. RFC 2598, Internet Request For Comments. June 1999.

[2] POSIX 1003.1b-1993. Portable Operating System Interface (POSIX) Part 1: System Application Program Interface Amendment 1: Realtime Extension [C Language], 1993.

[3] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels and John S. Quarterman. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading, MA, 1996.

[4] David Clark. The Design Philosophy of the DARPA Internet Protocols. *Computer Communication Review*, Vol. 18, No. 4, 1988, pp. 106-114.

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

[6] Kenjiro Cho. A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers. *Proc. USENIX Annual Technical Conference*, New Orleans, LA, June 1998, pp. 247-258.

[7] Tatuya Jinmei, Kazu Yamamoto, Jun-ichiro Hagino, Munechika Sumikawa, Yoshinou Inoue, Kazushi Sugyo and Soichi Sakane. An Overview of the KAME Network Software: Design and Implementation of the Advanced Internetworking Platform. *Proc. 9th Annual Conference of the Internet Society (INET'99)*, San Jose, CA, USA, 1998.

[8] Jon Postel. DARPA Internet Protocol Specification. RFC 791, Internet Request For Comments. September 1981.

[9] David D. Clark and Wenjia Fang. Explicit Allocation of Best-Effort Packet Delivery Service. *IEEE/ACM Transactions on Networking*, Vol.6, August 1998, pp. 362-373.

[10] Alok Gupta, Dale O. Stahl and Andrew B. Whinston. Priority Pricing of Integrated Services Networks. In *Internet Economics*, L. W. McKnight and J. P. Bailey (editors), MIT Press, 1997, pp. 323-352.

[11] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Internet Request For Comments, December 1998.

[12] ATM Forum. ATM Forum Traffic Management Specification Version 4.1. AF-TM-0121.000, March 1999.

[13] Jan Thibodeau (editor). The Basic Guide to Frame Relay Networking. Frame Relay Forum, Fremont, CA, USA, 1998.

[14] Carl A. Waldspurger and William E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[15] Klara Nahrstedt and Jonathan M. Smith. Design, Implementation and Experiences with the OMEGA End-point Architecture. *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 7, September 1996, pp. 1263-1279.

[16] David K. Y. Yau and Simon S. Lam. Migrating Sockets – End System Support for Networking with Quality of Service Guarantees. *IEEE/ACM Transactions on Networking*, Vol. 6, No. 6, December 1988, pp. 700-716.

[17] K. Lakshman, Raj Yavatkar and Raphael Finkel. Integrated CPU and Network-I/O QoS Management in an Endsystem. *Computer Communications*, Vol. 21, No. 4, April 1998, pp. 325-333.

[18] Tarek F. Abdelzaher and Kang G. Shin. QoS Provisioning with qContracts in Web and Multimedia Servers. *Proc. 20th IEEE Real-Time Systems Symposium*, Phoenix, AZ, USA, December 1999, pp. 44-53.

[19] Chen Lee, Katsuhiko Yoshida, Cliff Mercer and Ragunathan Rajkumar. Predictable Communication Protocol Processing in Real-Time Mach. *Proc. IEEE Real-Time Technology and Applications Symposium*, June 1996, pp. 220 –229.

[20] Lars Eggert and John Heidemann. Application-Level Differentiated Services for Web Servers. *World Wide Web Journal*, Volume 3, Issue 2, 1999, pp. 133-142

[21] Jussara Almeida, Mihaela Dabu, Anand Manikutty and Pei Cao. Providing Differentiated Levels of Service in Web Content Hosting. *Proc. 1988 SIGMETRICS Workshop on Internet Server Performance*, Madison, WI, USA, June 1998, pp. 91-102.

[22] John Bruno, José Brustoloni, Eran Gabber, Banu Özden and Abraham Silberschatz. Retrofitting Quality of Service into a Time-Sharing Operating System. *Proc. USENIX 1999 Annual Technical Conference*, Monterey, CA, USA, June 1999, pp. 15-26.

[23] Peter Druschel and Gaurav Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. *Proc. 2nd USENIX Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, USA, October 1996, pp. 261-275.

[24] Joe Touch and Amy S. Hughes. The LSAM Proxy Cache - a Multicast Distributed Virtual Cache. *Computer Networks and ISDN Systems*, Vol. 30, No. 22-23, November 1998, pp. 2245-2252.