

netnice: nice is not only for CPUs

A Simple Subnetwork Bandwidth Management Scheme

Takashi Okumura Mark Moir* Daniel Mossé†
Department of Computer Science,
University of Pittsburgh, Pittsburgh, PA 15260
{taka, moir, mosse}@cs.pitt.edu

Abstract

In this paper, we present “Netnice”, a mechanism that allows processes to throttle their own network bandwidth consumption. As the name suggests, it is inspired by the Unix `nice` command in that it allows users and administrators to limit the network resources used by individual processes in order to avoid impacting the performance of other processes. In the paper, to address the problem of transient performance deterioration in local area network (LAN) environments, we propose the bandwidth limitation primitive that works within a host’s kernel, and show several uses of *netnice*. Through experimentation in a small LAN of FreeBSD machines, we show how *netnice* allows for higher degree of controllability in bandwidth management.

1 Introduction

Although bandwidth control has gained momentum lately in network research, it has remained relatively unexplored within LAN environments. For example, a common situation is a LAN with remote file servers and remote compute servers: a user can experience significant slow-down of interactive sessions (e.g., `telnet`), when other users are carrying out network-intensive operations, such as bulk file transfers.

This lack of bandwidth control within LANs is due to several factors. First, most LANs have historically had far higher bandwidth than WANs, although they work only over short distances. Second, thanks to the sales volume of the dominant standard, Ethernet, bandwidth resources of LANs have remained extremely cheap. Third, only recently have CPUs become fast enough to enable a single node to consume the entire bandwidth of a LAN. Lastly, it is simpler to control traffic going through a single node than to control bandwidth *within* a LAN that may have shared media and multiple points of control. Consequently, we still observe transient performance deterioration due to bursty traffic in a LAN.

The problem of shared resources management has been extensively studied in different contexts. For instance, some applications (e.g., image manipulation) can severely affect the response time of other processes. In Unix-like OSs, the standard tool to circumvent this problem is the `nice` command, which enables

users and system administrators to change (typically lowering) the priority of processes. With this mechanism, users can collaborate to achieve “good” performance of most jobs even in relatively loaded systems.

We observed that similar approaches can be used in controlling network bandwidth. Previous attempts to address this issue have concentrated on limiting bandwidth consumption on a per-flow basis. In contrast, our *netnice* approach gives users the ability to selectively limit the bandwidth consumption on a **per-process** basis.

Typical users are not conscious of their processes’ independent connections, but are used to the per-process interface. For example, a user is usually not aware of the multiple connections opened by a process (e.g., a web browser requesting a webpage containing pictures). Thus, in this paper, we introduce the concept of a *process bandwidth controller*, and present a mechanism to provide the user with a means to limit the aggregate bandwidth consumption of entire processes. A process’s traffic is regulated by packet shaping at the network layer, which is controlled by simple system calls invoked from the command line or from within applications.

The rest of this paper is organized as follows. In the next section, we review previous work related to bandwidth control. We present our *process bandwidth controller* model in Section 3. In Section 4, we describe our implementation in FreeBSD [7], which includes a process bandwidth controller mechanism within the kernel, system calls, and three convenient ways for users to limit bandwidth of client and/or server processes. In Section 5, our evaluation quantifies the bandwidth controllability of our own mechanism, and in Section 6 we evaluate a practical case (with interactive and background traffic).

2 Related Work

Our work has been inspired by previous attempts to make fair use of a resource, which has been extensively researched in the area of CPU scheduling. These efforts include the `nice` facility and more recently the approach for controlling resource utilization without modifying applications [1]. In this section, we review closely related work from the networking perspective.

2.1 Hop-by-hop control

Packet scheduling is a common approach for guaranteeing quality of service (QoS) in packet switching networks. Several

*Work supported in part by an NSF CAREER Award, CCR 9702767
†Supported in part by DARPA under contract DABT63-96-C-0044, through the FORTS project.

advanced packet scheduling disciplines have been proposed to maintain efficiency and improve service differentiation: Priority Queuing (PQ), Fair Queuing (FQ), Weighted Fair Queuing (WFQ)[3], and Class Based Queuing (CBQ)[6].

For QoS guarantee strategies, several works have attempted to provide applications with real-time performance guarantees [5, 18, 2, 4, 10, 16, 11, 8]. These schemes differ in the strategies they employ to enforce rate control, in their policies for servicing packets, and in the types of guarantees they provide. Among these, Stop-and-Go Queuing [8], Jitter-EDD [16], and V-net [5] provide some form of deterministic service guarantees, with real-time scheduling of packets inside the network and some policing of traffic at the network edges.

For IP, the bandwidth reservation protocol, RSVP [12], uses periodic refresh messages to keep the route pinned down for a certain interval of time (i.e., it is a soft-state protocol).

These methods are mostly directed to the regulation of time-sensitive WAN traffic, where control is done in specific hosts. On the other hand, our target environment is a LAN, which may comprise a shared medium, and may not have a single point of control.

2.2 End-to-end control

Several rate control techniques for packet output have been presented to regulate or smoothen bursty traffic patterns. Packets are enqueued into different queues and dequeued to be put back into the outgoing traffic flow by scheduling algorithms such as the leaky bucket algorithm. A typical implementation defines a traffic class by IP address, transport protocol, application protocol, and sometimes application content.

These mechanisms have typically been used for traffic control in internetworking, and have not been used to address the issue of bandwidth control and interactions with other sessions in the same LAN, which comprises a variety of network devices. Moreover, this type of control has been typically provided by hardware: traffic shaper boxes that regulate the traffic going through the machines. However, it is impractical to deploy shaper boxes throughout the network to limit bandwidth of subnetworks, for economic and reliability reasons.

Nonetheless, unlike the packet scheduling disciplines, these mechanisms can be used to tame bursty traffic in a LAN if provided as a software solution. Implementation examples include delaying each packet as well as spoofing the rate of TCP acknowledgment packets and limiting the window size in the TCP header.

2.3 Subnet Bandwidth Management

Some works have been done in the IETF effort of Integrated Services over Specific Link Layers—ISSLL [14]. The primary motivation for the subnet bandwidth management in Internet standardization is the provision of seamless end-to-end bandwidth reservation using RSVP [12]. Recent work in this area includes Subnet Bandwidth Management (SBM) [17] and the Controlled Load Ethernet Protocol (CLEP) [9], which are intended for guaranteeing QoS between LANs and WANs and bandwidth management in a LAN segment, respectively.

It is important to point out that much of the previous work in

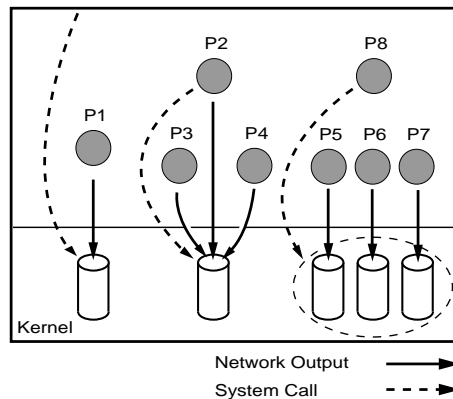


Figure 1: Process Bandwidth Controller Model.

this field has been done with *token-based* approaches for deterministic QoS, whereas our effort and the IETF's proposals are *tokenless* approaches. An example of a token-based approach is real-time Ethernet [15], which requires admission control. Recall that our goal is to improve network service quality in general without admission control.

3 Process Bandwidth Controller

3.1 Motivation

A typical LAN today accommodates various servers, such as file servers, databases, mail servers, web servers, news servers, and so forth, all of which can produce a high volume of data, either in bursts or over a prolonged period of time. Applications with modest bandwidth requirements may experience long delays due to these traffic patterns. Thus, the goal of *netnice* is to provide an efficient mechanism by which bandwidth consumption in LAN environments can be controlled dynamically by users or network administrators, in order to alleviate this problem. Benefits can be obtained by limiting the bandwidth consumption of only the applications that produce bursty or high-bandwidth traffic.

Operating systems already provide support for identifying individual *processes* and control their resource usage. For example, the Unix *nice* utility provides a way for users to dynamically change the priority of their processes. Analogously, by controlling the bandwidth consumption of individual processes, *netnice* can similarly provide a natural interface to users and network administrators.

3.2 Process Bandwidth Controller

For *per-process* traffic management, we employ a packet shaper for each process. Associated with each process is a *netnice value*, specifying the maximum bandwidth consumption of the process in bytes/sec. Users can control the process bandwidth consumption in several ways, as shown in Figure 1. The figure shows user processes denoted by circles, per-process packet shapers denoted by cylinders, bold arrows denote data flow from the processes, and dashed arrows denote control of the packet shaper attributes through *netnice* system calls (Section 4.2).

Process 1 (P1) in the figure is the simplest case. If the owner

of process P1 knows that P1 generates bursty traffic, s/he may reduce the output flow of P1 in order to avoid a detrimental effect on other processes.

The second example illustrates the *inheritance* of packet shaping among process families: P3 and P4 share a shaper with their parent process, P2. This is a useful characteristic of this model, since, we can make process families conform to the assigned bandwidth, regardless of the number of child processes they spawn (Note that by controlling the bandwidth of the `init` process, one can control the bandwidth of an entire server).

The third case shows an external *bandwidth manager* process, P8, assigning bandwidth among processes P5–P7. P8 may allocate available bandwidth fairly, or prioritize these processes based on request type, owner priority, etc. This scheme is particularly useful for bandwidth management of server machines, because an external agent can control the bandwidth consumption of server applications without modifying the application.

3.3 Discussion

The main disadvantage of `netnice` is that it does not provide minimum bandwidth guarantees, and thus, like `nice`, is not effective unless the owners of bandwidth-greedy processes voluntarily reduce their bandwidth consumption. Nevertheless, there are considerable advantages to this model, as outlined below.

First, users and system administrators can think about bandwidth management on a per-process basis, and not be concerned with individual flows, protocols, or interfaces. For example, a user can control the bandwidth consumption of a single web client or of several FTP sessions individually.

Second, it complies with the resource management model of underlying OSs. For example, once we set the `netnice` limit, the entire process family conforms to the limit, even if they spawn child processes or initiate new sessions. Flow-based traffic control primitives are able to emulate this behavior, but at extra cost.

Third, the *netnice* interface allows the bandwidth limitations assigned to processes to be modified dynamically. This is particularly useful because a user might not realize in advance that a process will be problematic. For example, if a web browser process suddenly starts to consume an excessive amount of bandwidth, thereby causing performance deterioration of a `telnet` session, the user (or a manager such as P8 in Figure 1) can introduce a bandwidth restriction on that web browser process.

Fourth, because a process's bandwidth consumption can be controlled externally, there is no need to modify applications. The per-process API we have developed allows bandwidth control to be performed in many ways, including (a) directly by the user using the `netnice` command, (b) by the shell, or (c) by a specialized bandwidth management process.

Finally, because *netnice* can limit the total bandwidth consumption of a process, we can achieve even receiver-side bandwidth control, provided TCP is used with the flow.

Note that even though *netnice* was originally intended for LAN bandwidth control, it also has useful applications in congestion avoidance of WAN traffic or in bandwidth control for resource conscious users, such as dialup users. Moreover, since *netnice* performs bandwidth control at the network level independently of the link layer, our approach is applicable to a wide

range of network devices, including wireless networks, and various wired topologies such as star and bus.

4 Implementation

For proof of the *netnice* concept, we implemented a packet shaping mechanism in the FreeBSD kernel, a couple of system calls to set and get the bandwidth specification of the shaper modules, and simple yet effective applications. The design and implementation of each component is briefly described in this section.

4.1 Kernel Module

We implemented an output rate control facility by putting a per-process shaping mechanism in the network output layer. Outgoing packets are added to the per-process queue. Instead of using a FIFO discipline, outgoing packets are scheduled to be released to the network as to limit the total bandwidth used by the corresponding process. The shaping mechanism monitors the traffic of each process, keeping statistics on the current bandwidth consumption in both directions. The statistics are used to meet the limitation of the total process bandwidth.

We chose to implement the traffic shaping mechanism in the network layer because locating it lower in the protocol stack does not meet our goal of regulating traffic on a per-process basis, rather than on a per-interface basis, and locating it higher in the protocol stack would require us to implement separate mechanisms for different transport protocols. Further, we decided against implementing a window-based mechanism in TCP since this mechanism would not be effective with short-lived sessions (e.g., the legacy HTTP GET operations).

In the following paragraphs, we give a brief description of components in the bandwidth regulation module.

Process Packet Shaper The packet shapers are implemented as a queue data structure with necessary statistics and timer, called `struct procpipe`. To access the `procpipe` easily, we added a pointer to the `procpipe` data structure from the process control block (PCB), which is called `struct proc` in BSD terminology.

The data structure is instantiated and the pointer is initialized when user sets the *netnice value* through system calls as described in Section 4.2. We chose this approach so that `procpipe` structures would be inherited by child processes in the same way that child processes inherit `nice` values.

Bandwidth Limitation Specification The per-process bandwidth limitation (*netnice value*) is expressed in bytes/sec in this implementation. For these values to be easily accessible from processes, we added a `value` variable to the PCB. Whenever `value` is updated, the `netnice` value is copied to the packet shaper structure, `procpipe`. This might be redundant, but it simplifies the system call implementation, and provides expandability. For example, we may want to allow users to change the effective limit by changing the value in `procpipe`, while enforcing the limit set by administrator in `proc`.

Packet Queuing Mechanism Our per-process packet queuing mechanism was implemented based on the packet preemp-

tion routine of Dummynet [13]. To enable efficient enqueueing of packets into the appropriate `struct procpipe` at the network layer, we added a pointer to the PCB (i.e., `struct proc`) into the packet data structure `mbuf`, which is initialized in the memory allocation routine for each packet.

Using the pointer, the queuing mechanism at the network layer can readily find the corresponding `procpipe` for an outgoing packet, without requiring any table lookup, which would be unacceptably costly on a server with heavy traffic.

Input Meter In a half-duplex environment, incoming packets share the media with outgoing packets. Thus, we need to monitor incoming traffic and take into consideration the bandwidth consumed by the incoming packets. To achieve this, we modified the `in_pcblookuphash()` function which is designed to deliver packets to the appropriate socket data structure at the destination node. Such modification allows for the monitoring of the incoming packets for each process without additional packet classification overhead.

Additionally, we provided a switch to enable and disable the input metering mechanism to meet specific needs of system administrators via the `sysctl` interface.

Packet scheduler We implemented the dequeuing of outgoing packets from the `procpipe` structure through a packet scheduler, which is invoked by a global software timer. The dequeued packets are returned to `ip_output()` at a constant rate; the rate is adjusted based on the monitoring mechanism described above, to comply with the `netnice` value specification. The packet scheduler sets the timer to go off at the scheduled time for the earliest packet in all active `procpipe` queues. Our packet scheduling granularity is 10 msec, for compatibility with the FreeBSD software timer.

4.2 System Calls

The `netnice` system call to get and set `netnice values` was implemented as an extension of `getpriority(2)` and `setpriority(2)`, like `nice`. We took the approach of extending existing system calls to minimize kernel modification:

```
int getpriority(int which, int who)
int setpriority(int which, int who, int
value)
```

which specifies the type of element the user wants to set or get; in our `netnice-enhanced` FreeBSD, we use `PRIO_NET` to set and get the `bandwidth specification`. `who` is the process id, and `value` is the `netnice value` of a process. For example, if we want to limit process 1234 to a maximum of 128Kbyte/sec, we invoke the `setpriority` system call as shown below.

```
setpriority(PRIO_NET, 1234, 128 * 1024);
```

4.3 Netnice User Interface

To realize the `netnice` concept depicted in Figure 1, we created several interfaces using the `netnice` API. The first scenario is achieved by the `netnice` command. The second case is illustrated by a `netnice` extension to the Bourne shell: the total bandwidth consumption of all the child processes spawned from the shell conforms to the bandwidth limit set for the shell by the extension. The third case, in which a bandwidth manager allocates

available bandwidth to several server processes, is realized as an extension to the existing network service super daemon, `inetd`. A concise explanation of each implementation is given below.

4.3.1 netnice command

We implemented the `netnice` command as a user command, which behaves like `nice(1)` and `renice(8)`. This command uses the system calls discussed above to configure a given process with the specified `netnice` value. The process traffic flow can be defined in terms of bandwidth and process id, as in the first example below. The second example returns the `netnice` value of process 1234. `netnice` with no parameters returns the `netnice value` of the shell being used.

```
% netnice 1234 128Kbyte/sec
```

```
% netnice 1234
```

Clearly, `netnice` allows the user or system administrator to dynamically throttle the network usage of processes.

4.3.2 Shell Extension

In addition to the commands above, we also provide an alternative way to limit the bandwidth at execution time. Using a simple bandwidth limitation symbol '@', users can limit bandwidth of their network-bound sessions. Because of the inheritance feature of `netnice`, the bandwidth limitation is the maximum aggregate bandwidth for a process and its children.

```
% ftp ftp.foo.com @128Kbyte/sec
```

This notation is also used in the following `inetd` examples.

4.3.3 Fair-inetd

This server application is intended to achieve fair allocation of network load among sessions, depicted in the third example of Figure 1. This `netnice-enhanced` `inetd` was implemented by putting a simple bandwidth calculation and allocation routine around the process start-up and termination modules. That is, near the `exec(2)` system call and the reaper routine.

Three sample configurations done through the configuration file for `inetd`, `/etc/inetd.conf`, and the server's behavior are described below (see Figure 2).

```
[A] ftp      ftpd      @128Kbyte/sec
     login    rlogind   @32Kbyte/sec
     telnet   telnetd   @32Kbyte/sec
```

```
[B] ftp      ftpd -1
     login    rlogind
     telnet   telnetd
```

```
[C] ftp      ftpd -1
     login    rlogind   @32Kbyte/sec
     telnet   telnetd   @32Kbyte/sec
```

Figure 2: Sample configurations of the `netnice-enhanced` `inetd`.

[A] In the first configuration, the value of each new process that is initiated by the daemon is specified. For example, each FTP connection is configured to have a 128Kbyte/sec limitation and each `telnet` and `login` session is given a 32Kbyte/sec limitation. Although this is not a firm bandwidth reservation,

telnet and *login* processes will probably get the requested bandwidth since the limitations target bandwidth-greedy protocols such as FTP.

[B] In the second configuration, the *inetd* will be assigned a limit (*inetd @512Kbyte/sec*). In this case, it will dynamically update the *netnice* values of processes, and each child process will receive $1/n$ of the total bandwidth, where n is the number of sessions. For example, if there are 4 concurrent FTP sessions, each session gets a 128Kbyte/sec limitation ($\frac{512K}{4} = 128K$).

[C] A combination of the first two configurations can be seen at the bottom of Figure 2. In this example, each *telnet* and *rlogin* session is limited to 32Kbyte/sec, while limits for the FTP sessions will be assigned dynamically to meet the total bandwidth. For example, suppose that there are 2 *telnet* sessions and 4 FTP sessions, and the super server has 512Kbyte/sec limitation. In this case, the bandwidth limits are: 32Kbyte/sec for each *telnet* session, and 112Kbyte/sec for each FTP session ($\frac{512 - (32+32)}{4} = 112$). As the number of sessions increases, the bandwidth assigned to each FTP process decreases accordingly.

5 Evaluation of the Primitive

In this section, we evaluate our implementation, and show that *netnice* controls traffic in LANs effectively and efficiently.

First, the overhead is measured by comparing the performance with a unmodified kernel. Next, we show how *netnice* is able to regulate the massive outgoing TCP traffic, as well as incoming traffic. Then, we demonstrate that *netnice* is an effective dynamic controller of bandwidth, through the *fair-inetd* introduced above.

In these experiments, all the machines have the same specification: Intel Celeron 433MHz, FreeBSD-2.2.8 OS, 96MB of memory, and 3Com EtherLink XL 905B 10/100 NIC.

5.1 Overhead in forwarding path

To evaluate the overhead caused by the *netnice* module, we measured the maximum throughput through a loop-back interface in the server machine (i.e., without using the physical network; the packets are just delivered to the local machine). This experiment was carried out with no bandwidth limitation, which is achieved by using a special *netnice* value of 0. We are measuring the impact our kernel modifications have on traffic. We measured throughput by transferring a 45Mbyte file, using FTP. The average of the results from 20 runs are shown in Table 1; these results suggest that no overhead is introduced by our implementation (the 2% difference is statistically insignificant).

5.2 Compulsory Bandwidth Regulation

Next we conducted two experiments to evaluate the bandwidth regulation mechanism. In these experiments, we started a process transferring a large amount of data between two machines, and then limited the bandwidth of the process using *netnice*. The first experiment studies output regulation (*netnice* is in the

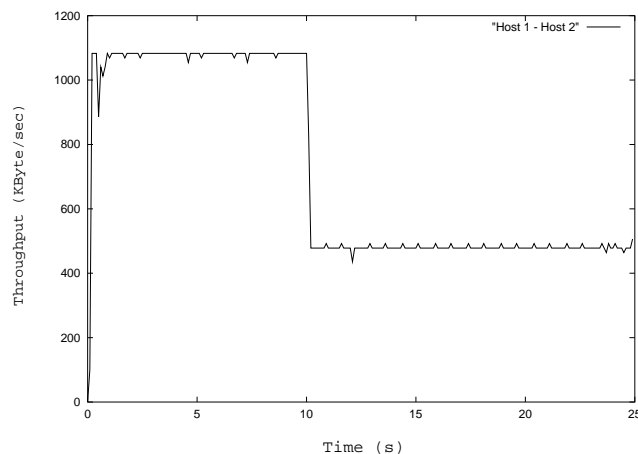


Figure 3: TCP Output regulation.

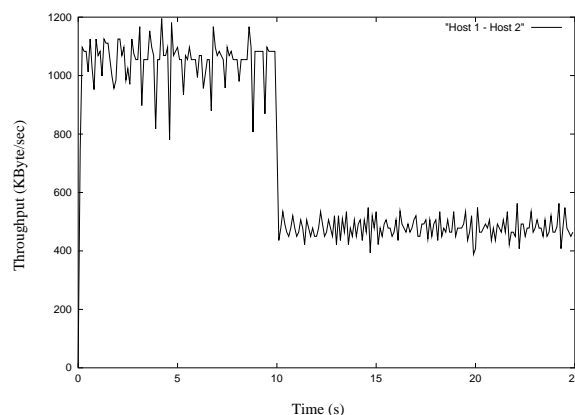


Figure 4: TCP Input regulation.

sender side; see Figure 3), and the second is aimed at evaluating input regulation (see Figure 4). To measure the traffic, we used the *tcpdump* command running on an additional client machine that was not involved in any of the experimental workloads, but was connected to the same 10Mbps Ethernet segment. In both experiments, the first 10 seconds shown are without any *netnice* limitation. Then, *netnice* was invoked to limit the bandwidth to 512Kbyte/sec.

The receive-side regulation is done by exploring the acknowledge packets that are generated by TCP. When there is heavy traffic, the release of acknowledge packets are scheduled taking into consideration the monitoring of incoming packets. In effect, the acknowledgement packets are delayed, which in turn indirectly slows down the sender. The receive side graph is noisier because *netnice* is applied this indirect way. The results show

	with netnice	unmodified
Throughput	13.94MBps	13.65MBps
Standard Deviation	0.66	0.58

Table 1: Overhead of *netnice*-enhanced kernel; throughput is measured with loop-back interface and, therefore, is independent from the network used.

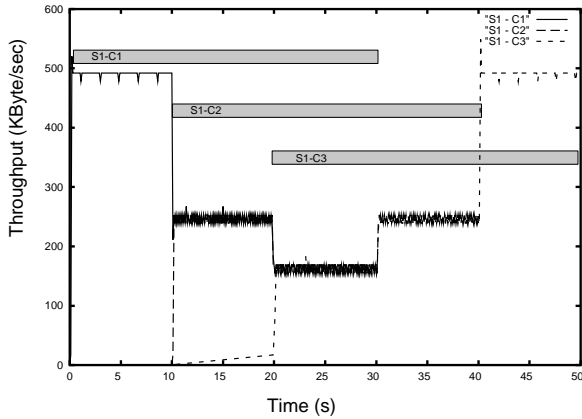


Figure 5: Total bandwidth limitation.

that *netnice* effectively limits both outgoing and incoming traffic.

5.3 Dynamic Bandwidth Regulation

In the next experiment, we measured the effects of the fair bandwidth allocation algorithm of the *inetd*, as shown in Figure 5. We used the third configuration of *inetd* shown in Figure 2, where the server has 512Kbyte/sec total bandwidth limitation.

Under this configuration, three concurrent FTP sessions were made, as shown by the gray rectangles in the figure, to see if the server conformed to the limitation. Each client, C1, C2, and C3, uses FTP to request a file from S1 at times 0, 10, and 20 seconds respectively, each lasting 30 seconds. Therefore, we have a single session from time 0 through 10 seconds and from 40 through 50 seconds, two concurrent sessions from 10 through 20 seconds and from 30 through 40 seconds, and three concurrent sessions from 20 through 30 seconds. These transfers are initiated and aborted automatically by a timer.

The figure clearly shows that *netnice* is effective in dividing the available bandwidth fairly amongst the active sessions, subject to the total bandwidth limitation of 512Kbyte/sec. All 512Kbytes/sec are assigned to the FTP session of C1 until 10 seconds, at which time another session (C2) becomes active. *Netnice* quickly adapts so that C1 and C2 get half of this bandwidth each, until their shares are further reduced (to a third of the total bandwidth) by the arrival of C3 at 20 seconds.

As shown, *netnice* worked quickly and effectively. The overhead measurement suggested its efficiency. Below we present practical scenarios in which users can perceive significant performance degradation due to a lack of bandwidth control in a LAN, and we show how *netnice* can improve the network performance.

6 A Sample Case with telnet

Heavy network traffic often adversely affects the performance of interactive applications, even when such applications have very little demand for network resources. In this section, we present experiments designed to demonstrate this phenomenon and to show that the *netnice* mechanism can be used to effectively

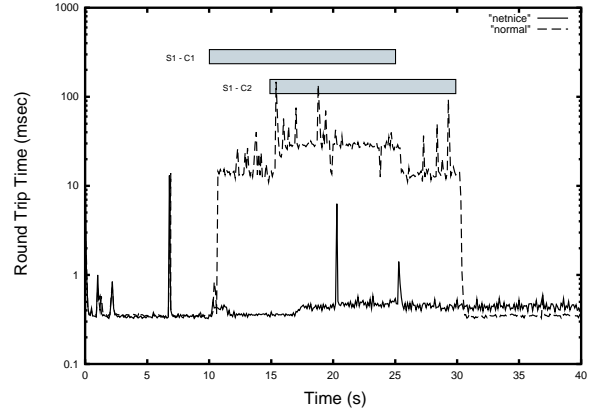


Figure 6: 10Base-T with one server.

eliminate degradation, provided the source of the traffic problem is only the server processes.

We created several representative LAN configurations and in each one we generated heavy file transfer traffic, while measuring round trip time (RTT) of packets from a *telnet* traffic emulator running concurrently with the file transfers. The *telnet* emulator sends one byte every 100 msec (i.e., at typical human typing speeds) using the TCP echo protocol. The byte is echoed back by the server and we measure the round trip time of each “key stroke” at the application level. With this simple, but effective tool, we show that our implementation greatly improves performance in several typical LAN configurations with bursty servers. As a workload, we transferred 128 Mbytes of data, with no disk access because disk access greatly influences the results.

All data shown below are from congested periods in the run, averaged over 10 experiments each.

6.1 10Base-T with one server

In this experiment, one FTP client in each of machines C1 and C2 request files from server S1, at times 10 seconds and 15 seconds, each lasting 15 seconds. The requests are handled by *fairinetd*, which is allocated 512Kbyte/sec total bandwidth. Meanwhile, the interactive traffic generator on machine C3 is sending “key strokes” to and getting its echo from S1. All machines are connected to a single 10Base-T hub. C3 is sending the simulated *telnet* traffic to S1 from time 0 through 40 seconds. Figure 6 shows the *telnet* RTT results for one experiment, although the other nine experiments yielded very similar results.

The RTT values for the unmodified kernel (i.e., without *netnice*) are between 10 and 150 ms, indicating at least an order of magnitude in performance deterioration for the *telnet* session. The average RTT for the unmodified configuration in the congested period, that is, from 10 through 30 seconds, was 23.05 msec, and the standard deviation (SD) was 31.72 msec.

In contrast, as can be seen from the graph, in the *netnice* configuration, spikes still exist, but network service quality is consistently good. In particular, note the spikes at times 1, 2, 7, 20 and 25 seconds. The first two are a result of the dispatching of the FTP scripts at 1 and 2 seconds. The other high values are due to some system activity, independent from the network load; this can be seen from the spike at time 7, which occurs

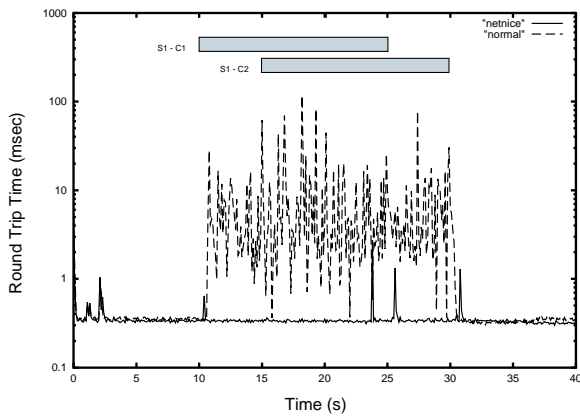


Figure 7: 10Base-T with two servers.

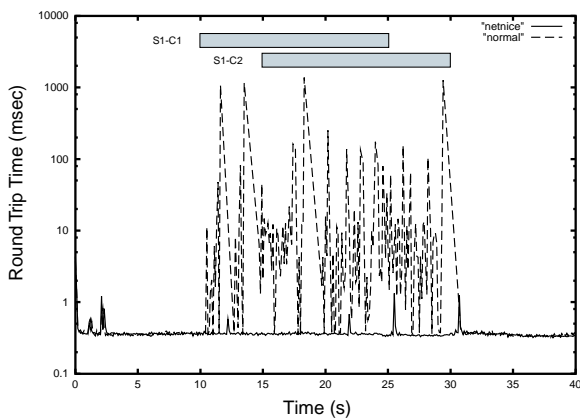


Figure 8: Switching Environment.

before the start of the file transfer. This behavior was observed in several of our experiments.

The average RTT for the *netnice* configuration was 0.49 msec, and SD was 0.97 msec. Overall, the performance of the interactive session is substantially improved by the use of *netnice*.

6.2 10Base-T with two servers

It is conceivable that the large delays for the unmodified kernel in the previous experiment are due to a bottleneck in the server or its interface. In this experiment, we added a server, S2, to the configuration shown in Figure 6. S2 serves only the interactive session, eliminating the possibility that the performance deterioration of the *telnet* session in the previous experiment was due to a server bottleneck. Otherwise, the experiment is identical to the previous one.

The results can be seen in Figure 7. Similar behavior is observed in this configuration without *netnice*, suggesting that the network segment is congested by the greedy FTP sessions from time 10 through 30 seconds. The average RTT and SD were 9.19 msec and 16.75 msec, respectively. In the *netnice* configuration, consistently good service quality is again observed. Average RTT and SD were 0.37 msec and 0.45 msec, respectively.

6.3 Switching Environment

Observe that in the previous experiment, the Network Interface Card (NIC) in S2 still has to process and discard all file transfer packets, and therefore could be the bottleneck. In order to rule out this possibility, in this experiment, we used an Ethernet switch (SOHware, 5-port 10/100 auto-sense), and created a typical LAN configuration with a switched server cluster. S1 and S2 were connected to the switch at 100Mbps speed, whereas C1, C2, and C3 were connected to the switch sharing a 10Mbps Ethernet segment. Since the switch isolates S2 from the file transfer traffic, it is possible to figure out where the real bottleneck is. The workload was the same as above.

The results, which can be seen in Figure 8, show a marked difference from those of the previous experiment: the average RTT and SD for unmodified kernel were 49.16 msec and 204.5 msec, compared to 0.48 msec and 1.35 msec for the *netnice* configuration. This suggests that the lack of bandwidth control when *netnice* is not activated allows S1 to send a high volume of data to the switch (at 100Mbps). The switch buffers the data and delays the packets of S2 more than if both senders were contending for a slower network link.

In the experiments, *netnice* reduced both delay (average RTT) and jitter (standard deviation) for the interactive sessions. This was true whether this session was communicating with the same server that was running the high-bandwidth file transfers, or with another server on the same network segment. The improvement was substantial in the cases where machines were powerful enough to consume the entire bandwidth of the network.

6.4 Discussion

The main disadvantage of a scheme like *nice* or *netnice* is that it requires the cooperation of all users involved in the network. In other words, the scheme might be easily compromised in a shared LAN model by a single non-conforming user flooding the network with legitimate or spurious packets. The system administrator has the power to offset the damage done by such a user, since s/he is able to control the amount of bandwidth assigned to a specific machine at boot time. This can be accomplished by setting a bandwidth limit to the *init* process, disallowing users from sending packets exceeding that amount. Clearly, such a configuration is quite inflexible for a LAN.

Receiver-side bandwidth regulation is possible using TCP (or any other acknowledge-based protocol) since the input meter will account for all traffic to and from a particular process. In this fashion, the acknowledgement will be inserted in the *procpipe* and will only be sent after much of the traffic has been received. When this happens, the sender will throttle its own bandwidth, waiting for acknowledgement packets from the receiver. The requirement of ack-based protocols is not a strong restriction, since many of the high-bandwidth protocols such as FTP and HTTP do use TCP for their underlying transport layer.

Differences between *nice* and *netnice* include: (a) the former is used for work conserving scheduling, unlike the latter; (b) the former works within CPU scheduling and controls a single host, while the latter needs to collaborate with packet schedulers on different hosts to provide control over a network segment; and (c) the former works with relative specification,

whereas the latter currently works with absolute specifications.

7 Concluding Remarks

Probably due to relatively inexpensive LAN resources, the bandwidth management problem in a local area network has not been addressed often. However, when network resources become scarce (due to bursty traffic or very fast computers connected to the LAN), a bandwidth management scheme is needed to achieve stable quality in network service.

In this paper, we proposed a scheme for allowing users and network administrators to limit the bandwidth consumption of applications on a **per-process basis**. Clearly, *netnice* cannot solve all subnetwork bandwidth problems, just like the original *nice* mechanism cannot solve all CPU scheduling problems. Yet, both schemes work quite effectively in many situations, specially those in which users are cooperative and the system administrator has control over the LAN resources.

Two points are worth noting. First, we provided a per-process bandwidth management mechanism which has a natural interface and works efficiently. Second, the benefits were gained without modifying application programs, but with small changes to the existing kernel and administrative programs such as a command, a shell, or a daemon.

Having implemented the basic mechanisms required for per-process bandwidth control, our future work will build on this foundation by providing a wider variety of ways to use these mechanisms. Our next step will be to implement an alternative traffic specification, which expresses a relative priority of a process for bandwidth consumption rather than an absolute bandwidth limitation. The benefit of this approach is that users can specify that their processes are willing to consume less bandwidth in the event that other processes are experiencing performance problems due to network congestion, but will not have to conform to an absolute bandwidth limitation otherwise. This will require a protocol that monitors current bandwidth consumption, and dynamically controls the bandwidth limitations of processes for which relative bandwidth priorities have been specified. Also, the topology of the LAN needs to be taken into consideration. Lastly, a simple interface to RSVP will allow *netnice* to provide end-to-end control.

The beta version of the code is available from <http://www.cs.pitt.edu/NETNICE>.

Acknowledgments The packet preemption mechanism of this implementation greatly owes to the Dummynet code by Dr. Rizzo. Kamran Farshchi has contributed to the project in many discussions.

References

- [1] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The eclipse operating system: Providing quality of service via reservation domains. In *USENIX Annual Technical Conference*, June 1998.
- [2] D. Clark, S. Shenker, and L. Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network:

- Architecture and Mechanism. *Computer Communication Review*, 22(4):14 – 26, October 1992.
- [3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Computer Communication Review*, 19(4):1 – 12, September 1989.
 - [4] D. Ferrari and D. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE J. on Selected Areas in Comm*, 8(3):368 – 379, April 1990.
 - [5] B. Field, T.F. Znati, and D. Mossé. V-net : A framework for a versatile network architecture to support real-time communication performance guarantees. *IEEE Transactions on Computers*, 1999.
 - [6] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. on Networking*, 3(4):365 – 386, August 1995.
 - [7] The FreeBSD Project. <http://www.freebsd.org/>.
 - [8] J.S. Golestani. A Framing Strategy for Congestion Management. *IEEE J. on Selected Areas in Communications*, 9(7):1064 – 1077, September 1991.
 - [9] E. Horlait and M. Bouyer. *Internet-Draft: CLEP (Controlled Load Ethernet Protocol): Bandwidth Management and Reservation Protocol for Shared Media*. IETF, July '99.
 - [10] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate Controlled Servers for Very High-Speed Networks. In *GlobeCom 90*, pages 12 – 20, May 1990.
 - [11] A. A. Lazar and G. Pacifici. Control of Resources in Broadband Networks with Quality of Service Guarantees. *IEEE Communications Magazine*, 29(10):66 – 73, Oct 91.
 - [12] Ed. R. Braden. *RFC 2205: Resource ReSerVation Protocol (RSVP) – Version 1*. IETF, September 1997.
 - [13] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31 – 41, January 1997.
 - [14] IETF The Integrated Services over Specific Link Layers WG. <http://www.ietf.org/html.charters/issll-charter.html>.
 - [15] C. Venkatramani and T. Chiueh. Supporting real-time traffic on ethernet. In *IEEE Real-Time Symposium*, 1994.
 - [16] D. Verma, H. Zhang, and D. Ferrari. Delay Jitter Control for Real-Time Communication in a Packet Switching Network. In *TriComm '91*, pages 35 – 43, 1991.
 - [17] R. Yavatkar, D. Hoffman, Y. Bernet, and F. Baker. *Internet-Draft: SBM (Subnet Bandwidth Manager): A protocol for RSVP-based admission control over IEEE 802-style networks*. IETF, May 1999.
 - [18] L. Zhang. *A New Architecture for Packet Switching Network Protocols*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, July 1989.