nlimit: A New Voluntary Bandwidth Limiting Facility

Sid Bottoms Thede Loder Rick Wash

December 11, 2002

Abstract

This paper introduces nlimit, a new voluntary bandwidth limiting system for UNIX. nlimit allows users to voluntarily limit their own network bandwidth usage by setting per-socket, per-process or per-user limits. The limits are then enforced by the UNIX kernel by shaping the outgoing stream of packets. Our system is effective, easy to use, and provides a good method for controlling network resource usage.

1 Introduction

Increasingly, administrators and users of network services need the ability to manage their use of bandwidth. This management may require fine tuning at the per-user (policy based) or even per-socket level. Some web servers, file servers, and peer to peer file-sharing agents now include such capabilities, implemented as user-level libraries. Examples include Apache, Zeus (web server), proftpd, pureftpd, Gnut, and Morpheus.

Unfortunately, using the kernel for bandwidth management is not currently available. Modern TCP/IP stacks and standard system call interfaces do not provide a means, leaving each application designer to build their own.

At the same time, another a common use of the network, that of interactive sessions (telnet and ssh), has exposed limitations in standard TCP. Interactive sessions may become extremely sluggish (due to increased latency) on single and multi-user systems when the connection to the network is saturated by other network uses, often controlled from the same machine. This is frustrating to both home users and those on ISP-operated multi-user systems, where one user can ruin all of the other active sessions by starting a few simultaneous downloads.

To address the needs of resource managing applications and servers, and with the idea that the operating system is an ideal place to address resource management issues, we have designed a common solution: a kernel provided bandwidth management API for both voluntary and involuntary limiting of connection throughput.

1.1 Existing Solutions

There are several existing application independent solutions. The two most common types are user-level libraries linked at runtime and external traffic shaping, separate from a process. User-level libraries work by intercepting calls to the libc functions for read and write. Traffic shaping typically uses packet filtering to match TCP/IP streams and a kernel facility for changing flow rates. While in some cases effective, each of these has a number of drawbacks:

User Level Libraries

- Fine grained timing available in the kernel and tcp/ip stack itself are unavailable to user programs
- Network resources are inefficiently used (large buffers are allocated in the kernel for fast flows, but may be unnecessary)
- Increased user and/or programmer work to setup

- Lack of clear winner (non standardized approaches, multiple implementations)

Traffic Shaping

- Existing bandwidth limiting is on a per IP address/port or per interface basis, and must be configured ahead of time
- Management of shaper requires root access
- Inaccessible to user programs (except via convoluted scripts)
- No policy level control for users, processes, or sockets
- No way to limit on a per connection basis unless all connections are symmetric.

The layout of the paper is as follows. In Section 2, we discuss related work. In Section 3, we describe the API and the userland tools that are used to access our system. Section 4 describes the design and implementation of the kernel portion of our system, which is where the limiting actually happens. Then in Section 5 describes our evaluation of the effectiveness of our solution. Section 6 discusses some possibilities for future work. Finally Section 8 gives our conclusions.

2 Related Work

2.1 ALTQ

ALTQ[2] is a traffic shaping scheme built into all versions of BSD. ALTQ allows the administrator to set up different queuing disciplines. There are several different queuing schemes, including CBQ, HFSC, and RED. Although ALTQ's queuing schemes allow sophisticated queuing control, ALTQ only supports prioritization based on port numbers. Our solution provides finer control based on processes, users, and sockets. Another problem with the existing ALTQ interface is that it is set up on a semi-permanent basis using a configuration file; it was not designed for dynamic changes.

2.2 Class-Based Queuing (CBQ)

CBQ[3] is a hierarchical link-sharing algorithm that provides a mechanism for fairly dividing bandwidth between applications depending on their needs. Its ability to partition bandwidth consumers is an important aspect that we utilize. CBQ is a very powerful and flexible algorithm.This algorithm has the advantage of being able to guarantee each application a share of the bandwidth during congestion, while still allowing excess bandwidth to be utilized. We use it as the underlying discipline behind our system.

2.3 Netbrake

Netbrake[11] is a user-level program that allows bandwidth to be limited on a per-process basis. It wraps calls to `read()`,`write()`, `socket()`, and so on to achieve the desired effect. To limit a process's bandwidth, you run:

```
% netbrake <bpslimit> <programname>
```

Although simple to implement and use, netbrake is limited in several ways: it can't differentiate between sockets in the same program, and it has no mechanism for the process to limit its own bandwidth. In addition, the bandwidth limit of a process can't be changed once the process is running.

2.4 Wonder Shaper

Wonder Shaper[4] is a tool that uses the Linux CBQ system. It sets the total bandwidth usage to just under the actual maximum. The author found that this simple step dramatically decreased the latency for interactive applications. It turned out that much of the latency was being caused by overflowing queues in this modem, and that setting a slightly lower bandwidth limit allowed the queues to stay almost empty. Similar to Netbrake, the main difference between this tool and our solution is that this tool can only work with ports or an entire interface.

2.5 DummyNet

Originally designed as a research tool, DummyNet[5] is used to simulate networks. The user can specify a specific connection's speed, thereby limiting that interface's bandwidth. DummyNet has been used outside of the research world for bandwidth management. However, DummyNet was not intended to be a general purpose tool, and it implements several simulation functions that are not needed for a bandwidth-management system. Furthermore, DummyNet lacks the ability to limit bandwidth on a per-user or per-process basis.

2.6 Diffserv and Intserv

Diffserv[6], or differentiated services, is a QoS scheme for TCP/IP that uses "relative sensitivities" of traffic to delay and loss. This will not work for us, since out target applications require hard limits on bandwidth.

Intserv[7] is closer to our work, since it can attempt to set bandwidth to a certain level. However, the processing for bandwidth control is done on the network. The significant overhead from this processing is an unnecessary burden for the Internet routers, since bandwidth limiting can be done on the host end.

3 API Design

We have provided a useful application programming interface that allows a process to voluntarily limit its own bandwidth on a given socket, or process-wide. Also, it will allow another process to impose bandwidth limitations based on user ID or process ID. We have provided a small userland utility to expose some of this functionality to the user.

3.1 Socket API

The first part of the API is designed to allow a process to set a bandwidth limit on each of its own sockets individually. In order to achieve this, we allow a user to set bandwidth limit as a standard socket option. The `setsockopt(2)` system call interface is utilized as a simple but powerful API. This permits a separate and different limit on each socket. The socket API is summarized in Figure 1.

`setsockopt`(**socketfd, ..., TCP_RCVRATE, n**)
: Set the inbound bandwidth limit for this particular socket.

`setsockopt`(**socketfd, ..., TCP_SNDRATE, n**)
: Set the outbound bandwidth limit for this particular socket.

`getsockopt`(**socketfd, ..., TCP_RCVRATE, n**)
: Returns the current inbound bandwidth limit for this particular socket.

`getsockopt`(**socketfd, ..., TCP_SNDRATE, n**)
: Returns the current outbound bandwidth limit for this particular socket.

Figure 1: Socket API Summary

The API supports setting any limit from 1 byte per second to 1 gigabyte per second. The API also supports changing the bandwidth limit in the middle of a connection as often as needed. Setting the bandwidth limit to 0 removes the limit from the socket.

At the moment, nlimit does not support limiting downloads. As such, attempting to set the receive bandwidth rate will return an error.

3.2 Device API

The rest of the `nlimit` API consists of a new device, `/dev/altq/nlimit`. This device is accessed solely through the `ioctl(2)` system call. `nlimit` has 4 different calls that can be made. Through this device it is possible to set a limit on all sockets of a given process, or all sockets created by a given user. A summary of this API is available in Figure 2.

In setting a process limit, the limit applies to every socket in the process such that the total bandwidth used by all of the sockets does not ex-

`ioctl`(`/dev/altq/nlimit`,
NLIMIT_SETLIMIT_USER, uid, n)
Set the upload bandwidth limit for all sockets/processes owned by a given uid. The sum total of all bandwidth used by this user will not exceed n.

`ioctl`(`/dev/altq/nlimit`,
NLIMIT_GETLIMIT_USER, uid)
Return the bandwidth limits (both upload and download) for the given user.

`ioctl`(`/dev/altq/nlimit`,
NLIMIT_SETLIMIT_PROCESS, pid,n)
Set the upload bandwidth limit for all sockets in a given process. The sum total of all bandwidth used by this process will not exceed n.

`ioctl`(`/dev/altq/nlimit`,
NLIMIT_GETLIMIT_PROCESS , pid)
Return the bandwidth limits (both upload and download) for the given process.

Figure 2: Device API Summary

ceed the process bandwidth limit. Each individual socket can use as much bandwidth as it likes (unless they are also limited), but they will compete for the bandwidth available for the process. This limit only applies to sockets that are created by this process, not to sockets that it receives via `fork(2)` or via file descriptor passing.

A process limit can only be set for a currently running process, and is automatically destroyed when the process terminates. Also, root can set a process limit on any process in the system, but a normal user can only set process limits on processes that are running as their user ID.

User bandwidth limits allow a user to set a maximum bandwidth used by all of the user's processes and sockets. This is similar to system resource limits on CPU time and processes. When set, the sum of all the bandwidth used by all sockets owned by the user must not exceed the user bandwidth limit.

A user limit can be set at any time, and is never automatically removed. Currently, root can set a user limit on any user (including root), and normal users can only set a limit on themselves. Also, there is no enforcement on limits set by root, so a normal user can remove a limit that was set by root.

3.3 nlimit Utility

Finally, a userland utility was written to permit easy access to this API. The utility, called `nlimit`, gives the ability to set or remove user or process limits from the command line. This utility has a fairly simple interface:

```
nlimit [-p pid | -u user] -o limit
```

To set a limit, specify either a user or a process to limit, and the limit that is to be set. Setting a limit of 0 removes the bandwidth limit. This provides a simple way to limit a connection of any process running on the machine, even processes that don't have support for `nlimit`.

4 Kernel Design

nlimit leverages the AltQ[2, 8] packet queuing and shaping framework as it exists in NetBSD[9] 1.6.

Currently, the existing AltQ framework has hooks into the TCP/IP stack. Whenever the TCP/IP stack wishes to output a packet, it makes a call into AltQ's Enqueue operation. Enqueue then classifies the packet and puts it onto a queue according to the specified queuing discipline. Then, at various points in time (such as when the network interface has open buffer space) the network interface will call AltQ's Dequeue operation to retrieve a packet. All of the network interface drivers include the proper AltQ hooks in NetBSD 1.6. These queues are currently only for sending data.

The behavior of these two operations is dependent on the pluggable queuing discipline used by AltQ. Each discipline can make its own choices on which packets from the queue to return to the

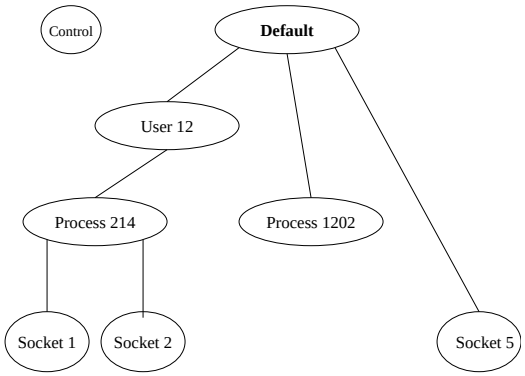

Figure 3: Class Hierarchy

network interface driver and when to return them. This allows different scheduling and quality-of-service algorithms to be dropped into the system and chosen between easily.

`nlimit` includes a new queuing discipline for AltQ that regulates data on the basis of the originating user, process, and/or socket. Unfortunately, AltQ does not currently permit classifying and queuing on this information. We modified other parts of the kernel pass necessary information into the AltQ system.

4.1 Queuing Discipline

First is our queuing discipline. Named `nlimit`, it is derivative of the Class-based Queuing system[3]. It uses the same mechanisms as CBQ to actually limit the packets, but nlimit manages a custom hierarchy of classes to achieve its goals.

4.1.1 Class Hierarchy

At the root of the hierarchy is a default class. All packets that don't match any class will automatically be put into this class. (The only exception is the small control class which reserves bandwidth for important control protocols such as `ICMP` and `IGMP`.) Each new limit that nlimit places will have its own class with a custom bandwidth limit on it. Directly under the default class is where all user-wide limit classes will be placed. All process-wide limits that are set will be made subclasses of the appropriate user limit class if it exists, and placed under the default class if it does not. All socket-specific limit classes will be placed as subclasses of their process's process-wide limit class if it exists, and if it does not will be placed under the creating user's class if it exists, and under the default class otherwise. Figure 3 shows a view of the hierarchy.

This hierarchy allows the system to rate limit based on any combination of user ID, process ID, and socket, and all limits placed in the system will be enforced. Also, this allows all of a user's processes to compete for the bandwidth that a user has, but to not go over that bandwidth. A similar situation exists for sockets in a process.

Much work went into properly managing this hierarchy, giving the ability to insert and remove classes from arbitrary places in the hierarchy. Moving classes around when adding or deleting a class that has multiple children required some careful management.

4.1.2 Limit Operations

There are three possible actions that can happen on a limit. A new limit can be created, an existing limit can be updated, and a limit can be destroyed.

When a new limit is created, three things must happen. First, a class corresponding to this limit must be created. The class contains all of the data necessary to exact the bandwidth limit (such as packet queue data structures, etc.). This class computes a "nanoseconds per byte" value which is used to perform the actual bandwidth limiting. Second, this class must be inserted into the proper place in the class hierarchy described in section 4.1.1.

Finally, a new filter data structure must be created. This filter allows a packet classifier to associate every packet that is to be sent with a class that contains the bandwidth-limited queue for that packet. If the packet does not match any filters, then it gets associated with the unlimited default class.

Updating an existing limit is a straightforward

process. A new "nanoseconds per byte" value is computed and stored in the class data structure. From this point forward, the class will perform all calculations based on this new value, and the bandwidth will be limited to the new limit.

Destroying a limit is slightly more difficult. When a limit is deleted, the class data structure and all associated filters must be removed and freed. Also, the parent class must be updated to know that it has lost a child. And any children of the deleted class must also be moved to now be children of the deleted class's parent.

Currently, removing a class that has children classes causes a race condition kernel panic, so that has been disabled in the current version. We hope to correct this problem in the future.

4.1.3 Packet Classification

There are three types of filters that `nlimit` supports. First is one that will match only a specific socket on the machine. `nlimit` currently supports limiting TCP sockets, so the 5-tuple (source IP, source port, destination IP, destination port, protocol number) is unique for every socket on the machine. As such we create a filter that contains this 5-tuple of information, and can easily match this for every packet it sees. The second type of filter is a process filter. This filter simply contains a process ID to match. The third type of filter is a user filter which contains a user ID to match.

Now that we have a set of filters that can associate packets to classes, we have to create a new classifier that can perform this association. The filter for the first type of filter, the socket filter, is fairly simple. We look in the packet for the 5-tuple of the information we need. Then we look through our list of socket filters and see if any of them have the same 5-tuple of information.

Next are user and process filters. These are more difficult to match since each packet does not directly contain this information. What we can do, however, is lookup the TCP control block structure that the packet came from. The TCP control block stores all of the information needed to manage a TCP connection. We store the user id and the process id of each socket in its TCP control block structure. Now we have enough information to look through the list of user and process filters and see if any of them match the packet.

Finally, now that we have up to four classes which match this packet (three filter matches plus the default class). We look at the limits and see which limit is the slowest limit of the three and assign the packet to that class. We choose the slowest limit among the set to insure that a class can never go over any limit in its hierarchy.

Currently, our classifier performs all of these operations for every packet. This unfortunately has some CPU overhead on the network stack and can limit the maximum throughput a little bit. For future work we plan on implementing some optimizations to this system, such as caching the searched-for TCP control block structures and ascending the hierarchy of classes to find parent classes (since all matches must be in a vertical chain in the hierarchy).

4.2 Event Traps

For `nlimit` to function properly, some specific kernel events must be trapped and have hooks added to them. The two in particular that must be trapped are socket closing and process exiting.

The closing of a socket must be trapped so that we can destroy any limits that are associated with the socket. This is difficult because a socket is not truly closed when a process exits or a process calls `close(2)` on the socket. This is because TCP might need some extra time to send all of the data in the queue and to receive any data in transit afterwards. But finally, once all of that has been done, the socket is destroyed, and when this happens any limit associated with this socket is also destroyed.

The termination of a process must also be trapped in order to clean up any process limits associated with that process. We don't want the limits to hang around and then end up limiting some other unfortunate process that was unlucky enough to get the same PID.

In addition to these two traps, we also had to

trap the TCP connection phase. Since all the parts of the socket 5-tuple are not assigned until after the connection has been established, we do not actually create the limit until just after the connection was established with the remote machine. (Just after we receive the SYN-ACK packet) This has the side affect that the initial SYN packet is not counted toward the bandwidth limit.

4.3 Data Structure Modifications

The only system data structure that we had to modify is the TCP control block structure. Four new fields were added to this structure to store `nlimit` data.

The first two new fields are to store the send and receive bandwidth limits for this socket. These two fields are necessary in the case that a user calls `setsockopt(2)` before the connection is established. The new limits are stored in the TCP control block, and after the connection is established they are used to create the new bandwidth limits.

The second two fields are the user ID and the process ID that initially opened the socket. These are used in multiple places. First of all, they are stored into the `nlimit` data structures to enable matching for user limits and process limits. Secondly, they are stored in all of the socket and process limit filters to make it easier to find which classes correspond to which users and processes.

This brings up another limitation of `nlimit`. The user ID and process ID are only stored at creation time. This means that a process that changes its user ID (via `setuid(2)`) will be limited under the original ID not the new ID. Likewise, if a socket file descriptor gets passed among processes (via file descriptor passing or `fork(2)`), only the limit corresponding to the process ID that opened the socket will apply.

5 Evaluation

For our evaluation, we wanted to verify that our modifications met the following conditions:

1. Our solution correctly limits bandwidth based on socket number, process id, and user id
2. The limiting should not be bursty
3. Limiting should be able to improve latency
4. The limiting should not introduce unacceptable performance decreases

5.1 Methodology

For all of our throughput and latency tests, we set up a simple server on a foreign host that acted as a sink for all data sent to it. We then sent data from our nlimit computer and measure the throughput on the network using tcpdump. In addition, the test programs spawned ping commands to measure the latency seen by interactive programs.

We tested nlimit on several different systems. One system consisted of a laptop connected over a 100 Mbps LAN to a single server. We also verified these tests using a cable modem with a 120 Kbps upload speed. We also performed some tests on a DSL line.

5.2 Verifying Functionality

To establish the functionality of our solution, we measured the throughput on an unlimited connection, and then measured the throughput with socket limiting enabled. As Figure 4 shows, our system successfully limited the bandwidth. The saturated connection displays a great deal of burstiness. The throughput on the limited connection, on the other hand, stays relatively constant.

5.2.1 Socket Limits

In a production systems, there would most likely be several limited sockets running at the same time. Therefor, it was important for us to look at the behavior of multiple limited sockets. Figure 5 shows the results from a test that forks off several connections, each with its bandwidth limited to 3 KBps. Although the individual sockets are

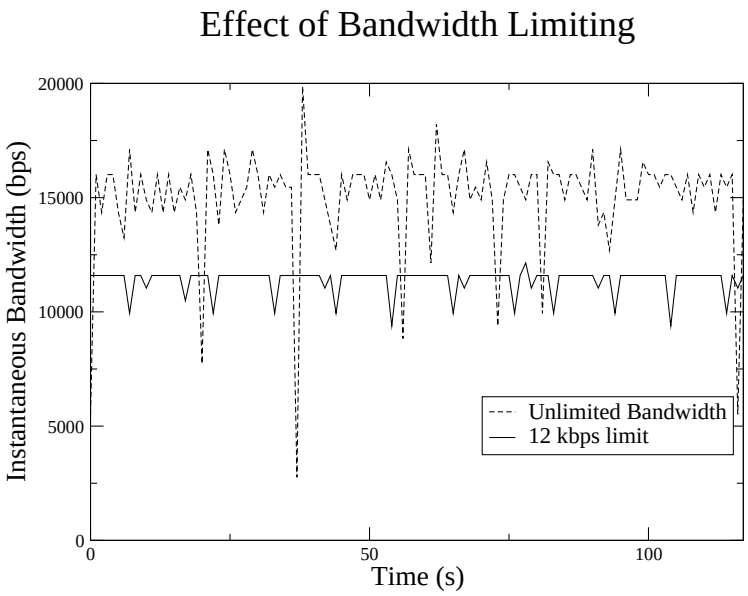

Figure 4: Effect of Bandwidth Limiting on Connection Bandwidth

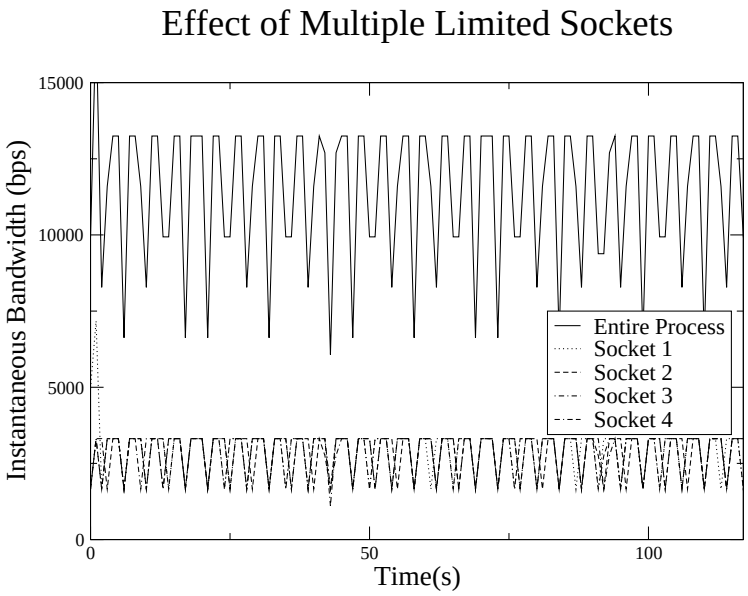

Figure 5: Multiple Limited Sockets

limited to approximately the right amount, they have a high degree of burstiness. The burstiness of the individual connections adds up to make the overall throughput very bursty.

We found that as the bandwidth limit approached the packet size, the transmission becomes more bursty. This happens because ALTQ limits throughput based on packets; it does not break up a packet to achieve the exact rate specified by the limit. This behavior is correct, since breaking the packets into smaller packets would interfere with the functionality of TCP. Moreover, the overall bandwidth averages out to be correct, and testing shows that we still achieve our goal of limiting the latency. A more thorough examination of the effects of small packet sizes on ALTQ/CBQ can be found in Russo[10].

5.2.2 Process and User Limits

In addition to testing the functionality of the socket-based limiting, we also tested the process and user-based limiting. Figure 6 shows our results for setting a process bandwidth limit on a multi-threaded program with four connections. This graph clearly shows the successful setting of the limit at 12 KBps. Individual connections compete for bandwidth in the same manner that they would if there was no limiting at all. Although not shown here, the bandwidth data for user-based limiting is essentially the same.

5.3 Low Bandwidth Connections

The usefulness of nlimit is determined to a large extent on what kind of connection your system has to the Internet. In the cable modem system, we saw massive gains in interactive performance when we turned on bandwidth limiting. This happens because the computer sends data to the cable modem much faster than the modem can send the data onto the network. The send queue on the cable modem then fills up and there is no room for interactive data to pass through. The DSL connection saw this behavior to some extent, but the higher upload speed of DSL means the increase in latency during the saturation tests wasn't as significant.

At the opposite end of the performance spectrum is the lab PC on a 100 Mbps connection across an empty network. In this situation, we can definitely limit the bandwidth as shown by our tests. However, we found that the latency remained relatively constant. This phenomenon arose because the computer is actually sending data just as fast as it can generate it.

5.4 Latency

To investigate the effects of rate limiting and competition between multiple simultaneous connections on latency, use again used the sink on a remote host across a WAN link.

Our test bed for this set of tests consisted of our modified NetBSD running as a guest operating system in VMWare Workstation on a Dell Inspirion 4000 laptop (256 MB, 100MB/sec Ethernet, 650 MHz Pentium) running Windows XP. The laptop was connected to a NAT-hidden isolated 100MB/sec LAN segment, and the NAT device was connected directly to a DSL line to the Internet. The DSL line had ISP imposed limits of 1.5 Mb/sec downstream and 768 kbps/sec upstream, respectively.

For each test using limiting, we started 1, 2, 4, or 8 nsend processes, each with a single `setsockopt()` limited socket. The bandwidth in each was set to $\frac{b}{n}$, where n is the total number of processes in the run and b is the total bandwidth. We used total bandwidths of 8KB/sec, 64KB/sec, 256KB/sec. 8K/sec was well within the maximum limit of the DSL line, 64KB/sec was within about 90%, and 256 KB/sec well beyond. We also ran the tests with no limiting.

Figure 7 shows these latency results. With a saturated link, the latency increases dramaticly with additional competing connections. Of particular interest are the results for 64KB/s aggregate throughput. This bandwidth is close to the saturation bandwidth of the link, yet even with 8 simultaneously connections, latency remains close to the minimum. Interactive sessions have am-

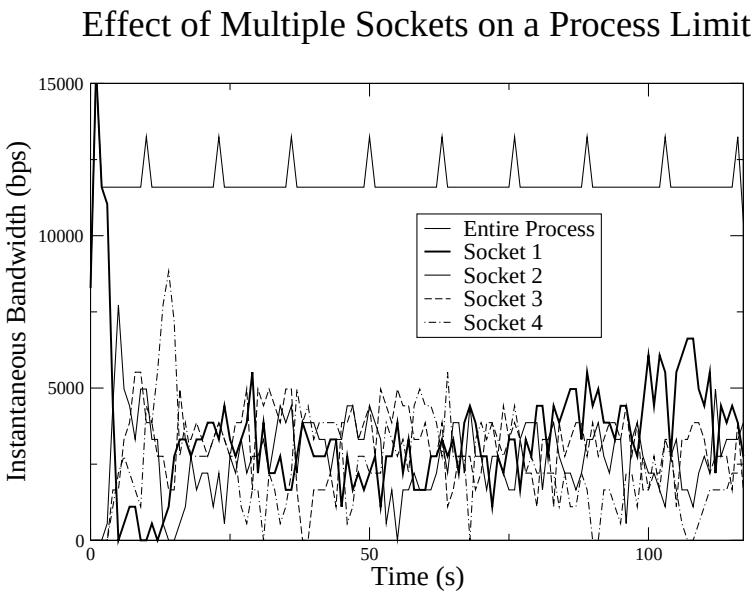

Figure 6: Multiple Unlimited Sockets Competing for Process Bandwidth

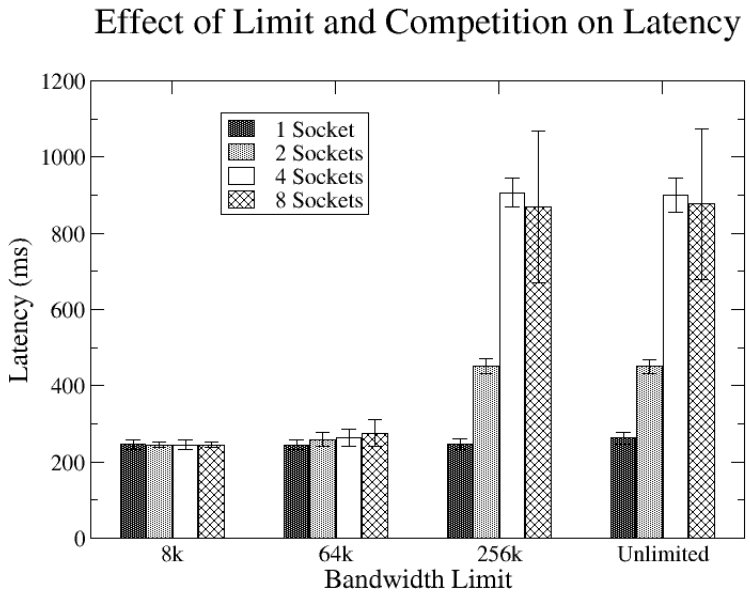

Figure 7: Effect of Limit and Competition on Latency

Kernel	Original	nlimit
nsend	7049000	2736000
slurp	6704000	3970000
socket-test	3461000	1966000

Table 1: Maximum Bandwidth (bytes per second) With and Without nlimit

Kernel	Average bps	Std-dev
Original	2.24	1.64
nlimit	1.311	0.231

Table 2: Latency Variation (ms) With and Without nlimit

ple remaining bandwidth to preserve low latency, demonstrating the benefit of voluntary limiting.

5.5 Performance

In order to evaluate the effects of adding nlimit to the networking stack in the kernel, we ran additional tests to test the maximum throughput. We compared the performance of an unmodified kernel to that of an nlimit enhanced kernel when transferring data across a 100 MB/sec isolated LAN segment. The results are presented in Table 1.

In this set of test, the 'slurp' program measured the effects of limits on upstream bandwidth when downloading a large file. It worked by connecting to the chargen port of inetd on the target machine, then reading the output as fast as possible. We checked the target machine to be sure that it was not the bounding performance factor.

The results show that nlimit introduces significant overhead. For instance the rate of transmission with nsend in the original kernel is 2 times faster than with nlimit. We suspect this is due to the much greater effort of classifying packets required by nlimit. Also, nlimit is not performance tuned. Section 4.1.3 discusses tuning strategies in more detail.

Latency, shown in Table 2, was less for the nlimit kernel than for for the original. This arises from ALTQ, which has the effect of normalizing the latency.

6 Future Work

There are a number of extensions to our existing `nlimit` infrastructure that would be interesting. First of all, it would be interesting to implement receive limits in AltQ and `nlimit`. Receive limits are non-trivial to get working properly and are currently unsupported.

There are a number of areas of our implementation that need work. First of all, our classifier should be much faster, since it needs to be called for every packet sent out over the network interface. Some methods of improving performance were discussed in Section 4.1.3.

Secondly, our implementation currently has a maximum bandwidth limit of 1 gigabyte per second. This is due to an overflow of a 32-bit integer. With network speeds increasing beyond this, it would be good to modify this to use a 64-bit integer to remove this limitation.

A third area that would be interesting to improve would be to properly handle sockets that get passed between processes or between users. Currently, a socket receives the limits of the process that created it, and the user that the process was running as when at creation time. Thanks to `fork(2)` and descriptor passing, it is possible to have a socket being used by multiple processes. It should then be subject to the process limits for both of these processes. Also, `setuid(2)` can cause a process to change who it is running as. This can be used as a way to get around user limits.

Another area that could use improvement is the permissions checking on user and process limits. The superuser should be able to set a maximum user bandwidth that the user cannot remove and have the user not be able to go over this bandwidth. This would make the user limits be along the lines of standard system resource limits. The same thing should be true for processes, in that root should be able to set a limit that the user cannot subsequently remove.

It should also be possible to extend the `altq.conf` syntax to include setting up some default bandwidth limits upon initialization. This would be useful to automatically limiting all scp session or automatically limiting all gtk-gnutella connections.

Finally, `nlimit` should be ported to other operating systems such as OpenBSD, FreeBSD, and Linux.

7 Use of the AltQ Framework

The use of the AltQ framework was a design decision that was made early in this project. It has proven to have many advantages, but also has been shown to have a number of disadvantages.

The use of AltQ allowed us to utilize the existing bandwidth limiting code saved us a lot of time and effort. AltQ already has hooks into the TCP stack and into all of the network device drives for sending and receiving packets. It also does all of the packet queue maintenance. This greatly simplified our implementation, as we could focus on setting and using the bandwidth limits.

Our use of the CBQ framework also simplified the project by allowing us, through the CBQ class hierarchy, to create bandwidth limits for users and processes in addition to sockets. Since CBQ was designed to manage multiple queues (one per class), this made the `nlimit` implementation simpler.

Unfortunately, using AltQ did have a number of unforeseen downsides. First of all, AltQ does not support setting receive bandwidth limits, only send bandwidth limits. Receive limits are a much more difficult problem to address since the kernel has little control over how much data is being sent by the remote machine. The only way to limit the receive bandwidth is to limit the amount of data that is acknowledged with TCP ACK's. Unfortunately, this is not an easy thing to do and AltQ does not support it. Since this support was not in AltQ, we decided not to include this support in our project. Our project has most of the hooks that would be needed if AltQ ever receives this support, though.

More in general, AltQ was mainly designed as a bandwidth shaping system for use on routers. The lack of receive support on a router does not matter, as it sends everything that is receives. Also, AltQ was designed with the idea that the set of classes and bandwidth limits would be setup at system boot time and stay mostly constant thereafter. `nlimit` adds dynamic changing of bandwidth limits, which caused us a number of debugging problems. In the end, AltQ handles dynamic changes in limits fairly well, but it took a number of workarounds to get it to manage this properly.

8 Conclusions

The `nlimit` kernel-based bandwidth limiting system provides a simple mechanism for programs and users to limit their bandwidth consumption. `nlimit` is effective at limiting the bandwidth in a number of common situations. It successfully limits connections, limits sockets in a process, and limits entire users. It also has an easy to use API and client program.

The benefits of `nlimit` increase in value when used on a slower network connection. Saturating the full capacity of a connection is easier on slow networks and is therefore more important to exercise control over the bandwidth usage of multiple programs.

References

[1] Cho, K. Framework for alternate queueing: Towards traffic management by pc-unix based routers. In *Proceedings of USENIX 1998 Annual Technical Conference* (June 1998).

[2] Cho, K. Managing traffic with alt. In *USENIX 1999 Annual Technical Conference: FREENIX Track* (June 1999).

[3] Floyd, S., and Jacobson, V. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions*

on Networking 3, 4 (August 1995). `http://www.icir.org/floyd/cbq.html/`.

[4] HUBERT, B. Wonder shaper. `http://lartc.org/wondershaper/`.

[5] HUBERT, B. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (January 1997). `http://citeseer.nj.nec.com/rizzo97dummynet.html`.

[6] INTERNET ENGINEERING TASK FORCE. *Diffserv*. `http://www.ietf.org/html.charters/diffserv-charter.html`.

[7] INTERNET ENGINEERING TASK FORCE. *Intserv*. `http://www.ietf.org/html.charters/intserv-charter.html`.

[8] KAME PROJECT. The kame project. `http://www.kame.net/`.

[9] NETBSD PROJECT. Netbsd. `http://www.netbsd.org`.

[10] RISSO, F., AND GEVROS, P. Operational and performance issues of a CBQ router. *ACM SIGCOMM Computer Communication Review 29*, 5 (October 1999).

[11] SANFILIPPO, S. Netbrake. `http://www.hping.org/netbrake/`.