STREAMS vs. Sockets Performance Comparison for UDP

Experimental Test Results for Linux

Brian F. G. Bidulock*
OpenSS7 Corporation

June 16, 2007

Abstract

With the objective of contrasting performance between STREAMS and legacy approaches to system facilities, a comparison is made between the tested performance of the *Linux Native Sockets* UDP implementation and STREAMS TPI UDP and XTIoS UDP implementations using the *Linux Fast-STREAMS* package [LfS].

1 Background

UNIX networking has a rich history. The TCP/IP protocol suite was first implemented by BBN using Sockets under a DARPA research project on 4.1aBSD and then incorporated by the CSRG into 4.2BSD [MBKQ97]. Lachmann and Associates (Legent) subsequently implemented one of the first TCP/IP protocol suite based on the Transport Provider Interface (TPI) [TLI92] and STREAMS [GC94]. Two other predominant TCP/IP implementations on STREAMS surfaced at about the same time: Wollongong and Mentat.

1.1 STREAMS

STREAMS is a facility first presented in a paper by Dennis M. Ritchie in 1984 [Rit84], originally implemented on 4.1BSD and later part of *Bell Laboratories Eighth Edition UNIX*, incorporated into *UNIX System V Release 3* and enhanced in *UNIX Sysvem V Release 4* and further in *UNIX System V Release 4.2*. STREAMS was used in SVR4 for terminal input-output, pseudo-terminals, pipes, named pipes (FIFOs), interprocess communication and networking. STREAMS was used in SVR3 for networking (with the NSU package). Since its release in *System V Release 3*, STREAMS has been implemented across a wide range of UNIX, UNIX-like and UNIX-based systems, making its implementation and use an ipso facto standard.

STREAMS is a facility that allows for a reconfigurable full duplex communications path, *Stream*, between a user process and a driver in the kernel. Kernel protocol modules can be pushed onto and popped from the *Stream* between the user process and driver. The *Stream* can be reconfigured in this way by a user process. The user process, neighbouring protocol modules and the driver communicate with each other using a message passing scheme. This permits a loose coupling between protocol modules, drivers and user processes, allowing a third-party and loadable kernel module approach to be taken toward the provisioning of protocol modules on platforms supporting STREAMS.

On *UNIX System V Release 4.2*, STREAMS was used for terminal input-output, pipes, FIFOs (named pipes), and network communications. Modern UNIX, UNIX-like and UNIX-based systems providing STREAMS normally support some degree of network communications using STREAMS; however, many do not support STREAMS-based pipe and FIFOs[1] or terminal input-output.[2]

UNIX System V Release 4.2 supported four Application Programmer Interfaces (APIs) for accessing the network communications facilities of the kernel:

Transport Layer Interface (TLI). *TLI* is an acronym for the *Transport Layer Interface* [TLI92]. The *TLI* was the non-standard interface provided by SVR4, later standardized by *X/Open* as the *XTI* described below. This interface is now deprecated.

X/Open Transport Interface (XTI). *XTI* is an acronym for the *X/Open Transport Interface* [XTI99]. The *X/Open Transport Interface* is a standardization of the *UNIX System V Release 4*, *Transport Layer Interface*. The interface consists of an Application Programming Interface implemented as a shared object library. The shared object library communicates with a transport provider Stream using a service primitive interface called the *Transport Provider Interface*.

While *XTI* was implemented directly over STREAMS devices supporting the *Transport Provider Interface (TPI)* [TPI99] under SVR4, several non-traditional approaches exist in implementation:

Berkeley Sockets. Sockets uses the BSD interface that was developed by BBN for TCP/IP protocol suite under DARPA contract on 4.1aBSD and released in 4.2BSD. BSD Sockets provides a set of primary API functions that are typically implemented as system calls. The BSD Sockets interface is non-standard and is now deprecated in favour of the POSIX/SUS standard Sockets interface.

POSIX Sockets. Sockets were standardized by the *OpenGroup* [OG] and *IEEE* in the POSIX standardization process. They appear in XNS 5.2 [XNS99], SUSv1 [SUS95], SUSv2 [SUS98] and SUSv3 [SUS03].

On systems traditionally supporting Sockets and then retrofitted to support STREAMS, there is one approach toward supporting *XTI* without refitting the entire networking stack:[3]

XTI over Sockets. Several implementations of STREAMS on UNIX utilize the concept of *TPI* over Sockets. Following this approach, a STREAMS pseudo-device driver is provided that hooks directly into internal socket system calls to implement the driver, and yet the networking stack remains fundamentally BSD in style.

Typically there are two approaches to implementing XTI on systems not supporting STREAMS:

XTI Compatibility Library. Several implementations of XTI on UNIX utilize the concept of an XTI compatibility library.[4] This is purely a shared object library approach to providing *XTI*. Under this approach it is possible to use the *XTI*

*bidulock@openss7.org

1. For example, AIX.
2. For example, HP-UX.
3. This approach is taken by True64 (Digital) UNIX.
4. One was even available for Linux at one point.

application programming interface, but it is not possible to utilize any of the STREAMS capabilities of an underlying *Transport Provider Interface (TPI)* stream.

TPI over Sockets. An alternate approach, taken by the *Linux iBCS* package was to provide a pseudo-transport provider using a legacy character device to present the appearance of a STREAMS transport provider.

Conversely, on systems supporting STREAMS, but not traditionally supporting Sockets (such as SVR4), there are four approaches toward supporting BSD and POSIX Sockets based on STREAMS:

Compatibility Library Under this approach, a compatibility library (`libsocket.o`) contains the socket calls as library functions that internally invoke the TLI or TPI interface to an underlying STREAMS transport provider. This is the approach originally taken by SVR4 [GC94], but this approach has subsequently been abandoned due to the difficulties regarding fork(2) and fundamental incompatibilities deriving from a library only approach.

Library and cooperating STREAMS module. Under this approach, a cooperating module, normally called `sockmod`, is pushed on a Transport Provider Interface (TPI) Stream. The library, normally called `socklib` or simply `socket`, and cooperating `sockmod` module provide the BBN or POSIX Socket API. [VS90] [Mar01]

Library and System Calls. Under this approach, the BSD or POSIX Sockets API is implemented as system calls with the sole exception of the `socket`(3) call. The underlying transport provider is still an *TPI*-based STREAMS transport provider, it is just that system calls instead of library calls are used to implement the interface. [Mar01]

System Calls. Under this approach, even the socket(3) call is moved into the kernel. Conversion between POSIX/BSD Sockets calls and TPI service primitives is performed completely within the kernel. The sock2path(5) configuration file is used to configure the mapping between STREAMS devices and socket types and domains [Mar01].

1.1.1 Standardization.

During the POSIX standardization process, networking and Sockets interfaces were given special treatment to ensure that both the legacy Sockets approach and the STREAMS approach to networking were compatible. POSIX has standardized both the XTI and Sockets programmatic interface to networking. STREAMS networking has been POSIX compliant for many years, BSD Sockets, POSIX Sockets, TLI and XTI interfaces, and were compliant in the *SVR4.2* release. The STREAMS networking provided by *Linux Fast-STREAMS* package provides POSIX compliant networking.

Therefore, any application utilizing a Socket or Stream in a POSIX compliant manner will also be compatible with STREAMS networking.[5]

1.2 Linux Fast-STREAMS

The first STREAMS package for Linux that provided SVR4 STREAMS capabilities was the *Linux STREAMS (LiS)* package originally available from GCOM [LiS]. This package exhibited incompatibilities with SVR 4.2 STREAMS and other STREAMS implementations, was buggy and performed very poorly on Linux. These difficulties prompted the OpenSS7 Project [SS7] to implement an SVR 4.2 STREAMS package from scratch, with the objective of production quality and high-performance, named *Linux Fast-STREAMS* [LfS].

The OpenSS7 Project also maintains public and internal versions of the *LiS* package. The last public release was *LiS-2.18.3*; the current internal release version is *LiS-2.18.6*. The current production public release of *Linux Fast-STREAMS* is *streams-0.9.3*.

2 Objective

The question has been asked whether there are performance differences between a purely BSD-style approach and a STREAMS approach to TCP/IP networking, cf. [RBD97]. However, there did not exist a system which permitted both approaches to be tested on the same operating system. *Linux Fast-STREAMS* running on the GNU/Linux operating system now permits this comparison to be made. The objective of the current study, therefore, was to determine whether, for the Linux operating system, a STREAMS-based approach to TCP/IP networking is a viable replacement for the BSD-style sockets approach provided by Linux, termed NET4.

When developing STREAMS, the authors oft times found that there were a number of preconceptions espoused by Linux advocates about both STREAMS and STREAMS-based networking, as follows:

- STREAMS is slow.
- STREAMS is more flexible, but less efficient [LML].
- STREAMS performs poorly on uniprocessor and ever poorer on SMP.
- STREAMS networking is slow.
- STREAMS networking is unnecessarily complex and cumbersome.

For example, the Linux kernel mailing list has this to say about STREAMS:

> **(REG)** STREAMS allow you to "push" filters onto a network stack. The idea is that you can have a very primitive network stream of data, and then "push" a filter ("module") that implements TCP/IP or whatever on top of that. Conceptually, this is very nice, as it allows clean separation of your protocol layers. Unfortunately, implementing STREAMS poses many performance problems. Some Unix STREAMS based server telnet implementations even ran the data up to user space and back down again to a pseudo-tty driver, which is very inefficient.
>
> STREAMS will **never** be available in the standard Linux kernel, it will remain a separate implementation with some add-on kernel support (that come with the STREAMS package). Linus and his networking gurus are unanimous in their decision to keep STREAMS out of the kernel. They have stated several times on the kernel list when this topic comes up that even optional support will not be included.
>
> **(REW, quoting Larry McVoy)** "It's too bad, I can see why some people think they are cool, but the performance cost - both on uniprocessors and even more so on SMP boxes - is way too high for STREAMS to ever get added to the Linux kernel."
>
> Please stop asking for them, we have agreement amoungst the head guy, the networking guys, and the fringe folks like myself that they aren't going in.
>
> **(REG, quoting Dave Grothe, the STREAMS guy)** STREAMS is a good framework for implementing complex and/or deep protocol stacks having nothing to do with TCP/IP, such as SNA. It trades some efficiency for flexibility. You may find the Linux STREAMS package (LiS) to be quite useful if you need to port protocol drivers from Solaris or UnixWare, as Caldera did.
>
> The Linux STREAMS (LiS) package is available for download if you want to use STREAMS for Linux. The following site also contains a dissenting view, which supports STREAMS.

The current study attempts to determine the validity of these preconceptions.

5. This compatibility is exemplified by the `netperf` program which does not distinguish between BSD or STREAMS based networking in their implementation or use.

3 Description

Three implementations are tested:

Linux Kernel UDP (`udp`*).*
: The native Linux socket and networking system.

OpenSS7 STREAMS XTIoS `inet` *Driver.*
: A STREAMS pseudo-device driver that communicates with a socket internal to the kernel.

  The OpenSS7 implementation of STREAMS XTI over Sockets implementation of UDP. While the implementation uses the Transport Provider Interface and STREAMS to communicate with the driver, internal to the driver a UDP Socket is opened and conversion between STREAMS and Sockets performed.

OpenSS7 STREAMS TPI UDP Driver `udp`*.*
: A STREAMS pseudo-device driver that fully implements UDP and communicates with the IP layer in the kernel.

The three implementations tested vary in their implementation details. These implementation details are described below.

3.1 Linux Kernel UDP

Normally, in BSD-style implementations of Sockets, Sockets is not merely the Application Programmer Interface, but also consists of a more general purpose network protocol stack implementation [MBKQ97], even though the mechanism is not used for more than TCP/IP networking. [GC94]

Although BSD networking implementations consist of a number of networking layers with soft interrupts used for each layer of the networking stack [MBKQ97], the Linux implementation, although based on the the BSD approach, tightly integrates the socket, protocol, IP and interface layers using specialized interfaces. Although roughly corresponding to the BSD stack as illustrated in *Figure 1*, the socket, protocol and interface layers in the BSD stack have well defined, general purpose interfaces applicable to a wider range of networking protocols.

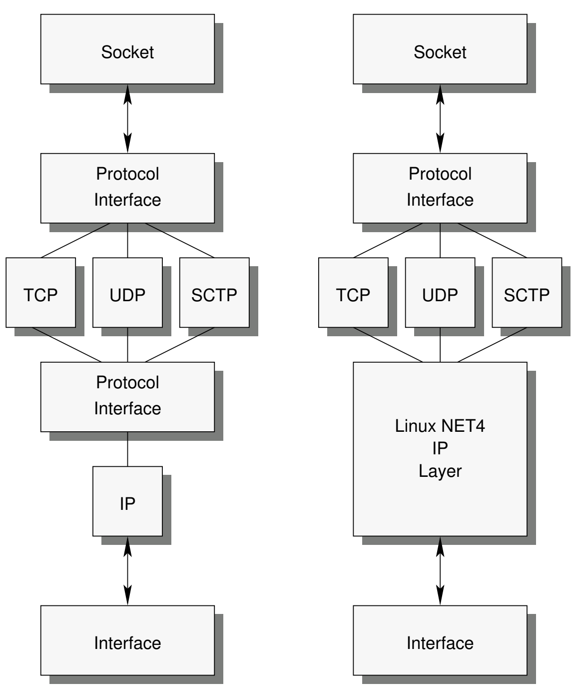

Figure 1: Sockets: BSD and Linux

Both Linux UDP implementations are a good example of the tight integration between the components of the Linux networking stack.

Write side processing. On the write side of the Socket, bytes are copied from the user into allocated socket buffers. Write side socket buffers are charged against the send buffer. Socket buffers are immediately dispatched to the IP layer for processing. When the IP layer (or a driver) consumes the socket buffer, it releases the amount of send buffer space that was charged for the send buffer. If there is insufficient space in the send buffer to accommodate the write, the calling processed is either blocked or the system call returns an error (`ENOBUFS`).

For loop-back operation, immediately sending the socket buffer to the IP layer has the additional ramification that the socket buffer is immediately struck from the send buffer and immediately added to the receive buffer on the receiving socket. Therefore, the size of the send buffer or the send low water mark, have no effect.

Read side processing. On the read side of the Socket, the network layer calls the protocol's receive function. The receive function checks if socket is locked (by a reading or writing user). If the socket is locked the socket buffer placed in the socket's backlog queue. The backlog queue can hold a maximum number of socket buffers. If this maximum is exceeded, the packet is dropped. If the socket is unlocked, and the socket buffer will fit in the socket's receive buffer, the socket buffer is charged against the receive buffer. If the socket buffer will not fit in the receive buffer, the socket buffer is dropped.

Read side processing under Linux does not differ from BSD, except for loop-back devices. Normally, for non-loop-back devices, `skbuff`s received by the device are queued against the IP layer and the IP layer software interrupt is raised. When the software interrupt runs, `skbuffs`s are delivered directly to the transport protocol layer without intermediate queueing [MBKQ97].

For loop-back operation, however, Linux skips queueing at the IP protocol layer (which does not exist as it does in BSD) and, instead, delivers `skbuff`s directly to the transport protocol.

Due to this difference between Linux and *BSD* on the read side, it is expected that performance results for Linux would vary from that of *BSD*, and the results of this testing would therefore not be directly applicable to *BSD*.

Buffering. Buffering at the Socket consist of a send buffer and low water mark and a receive buffer and low water mark. When the send buffer is consumed with outstanding messages, writing processes will either block or the system call will fail with an error (`ENOBUFS`). When the send buffer is full higher than the low water mark, a blocked writing process will not be awoken (regardless of whether the process is blocked in write or blocked in poll/select). The send low water mark for Linux is fixed at one-half of the send buffer.

It should be noted that for loop-back operation under Linux, the send buffering mechanism is effectively defeated.

When the receive buffer is consumed with outstanding messages, received messages will be discarded. This is in rather stark contrast to BSD where messages are effectively returned to the network layer when the socket receive buffer is full and the network layer can determine whether messages should be discarded or queued further [MBKQ97].

When there is no data in the receive buffer, the reading process will either block or return from the system call with an error (`ENOBUFS` again). When the receive buffer has fewer bytes of data in it than the low water mark, a blocked reading process will not be awoken (regardless of whether the process is blocked in write or blocked in poll/select). The receive low water mark for Linux is typically set to BSD default of 1 byte.[6]

6. The fact that Linux sets the receive low water mark to 1 byte is an indication that the buffering mechanism on the read side simply does not work.

It should be noted that the Linux buffering mechanism does not have hysteresis like that of STREAMS. When the amount of data in the send buffer exceeds the low water mark, poll will cease to return `POLLOUT`; when the receive buffer is less than the low water mark, poll will cease to return `POLLIN`.

Scheduling. Scheduling of processes and the buffering mechanism are closely related.

Writing processes for loop-back operation under UDP are allowed to spin wildly. Written data charged against the send buffer is immediately released when the loop-back interface is encountered and immediately delivered to the receiving socket (or discarded). If the writing process is writing data faster that the reading process is consuming it, the excess will simply be discarded, and no back-pressure signalled to the sending socket.

If receive buffer sizes are sufficiently large, the writing process will lose the processor on uniprocessor systems and the reading process scheduled before the buffer overflows; if they are not, the excess will be discarded. On multiprocessor systems, provided that the read operation takes less time than the write operation, the reading process will be able to keep pace with the writing process. If the receiving process is run with a very low priority, the writing process will always have the processor and a large percentage of the written messages will be discarded.

It should be noted that this is likely a Linux-specific deficiency as the BSD system introduces queueing, even on loop-back.

Reading processes for loop-back operation under UDP are awoken whenever a single byte is received (due to the default receive low water mark). If the reading process has higher priority than the writing process on uniprocessors, the reading process will be awoken for each message sent and the reading process will read that message before the writing process is permitted to write another. On SMP systems, because reading processes will likely have the socket locked while reading each message, backlog processing will likely be invoked.

3.2 Linux Fast-STREAMS

Linux Fast-STREAMS is an implementation of *SVR4.2 STREAMS* for the *GNU/Linux* system developed by the *OpenSS7 Project* [SS7] as a replacement for the buggy, underperforming and now deprecated *Linux STREAMS (LiS)* package. *Linux Fast-STREAMS* provides the STREAMS executive and interprocess communication facilities (pipes and FIFOs). Add-on packages provide compatibility between *Linux Fast-STREAMS* and other STREAMS implementations, a complete *XTI* shared object library, and transport providers. Transport providers for the TCP/IP suite consist of an `inet` driver that uses the *XTI over Sockets* approach as well as a full STREAMS implementation of SCTP (Stream Control Transmission Protocol), UDP (User Datagram Protocol) and RAWIP (Raw Internet Protocol).

3.2.1 XTI over Sockets

The XTI over Sockets implementation is the `inet` STREAMS driver developed by the *OpenSS7 Project* [SS7]. As illustrated in *Figure 2*, this driver is implemented as a STREAMS pseudo-device driver and uses STREAMS for passing TPI service primitives to and from upstream modules or the *Stream head*. Within the driver, data and other TPI service primitives are translated into kernel socket calls to a socket that was opened by the driver corresponding to the transport provider instance. Events received from this internal socket are also translated into transport provider service primitives and passed upstream.

Write side processing. Write side processing uses standard STREAMS flow control mechanisms as are described for TPI, below, with the exception that once the message blocks arrive at the driver they are passed to the internal socket. Therefore, a unique characteristic of the write side processing for the XTI over Sockets driver is that data is first copied from user space into STREAMS message blocks and then copied again from the STREAMS message blocks to the socket. This constitutes two copies per byte versus one copy per byte and has a significant impact on the performance of the driver at large message sizes.[7]

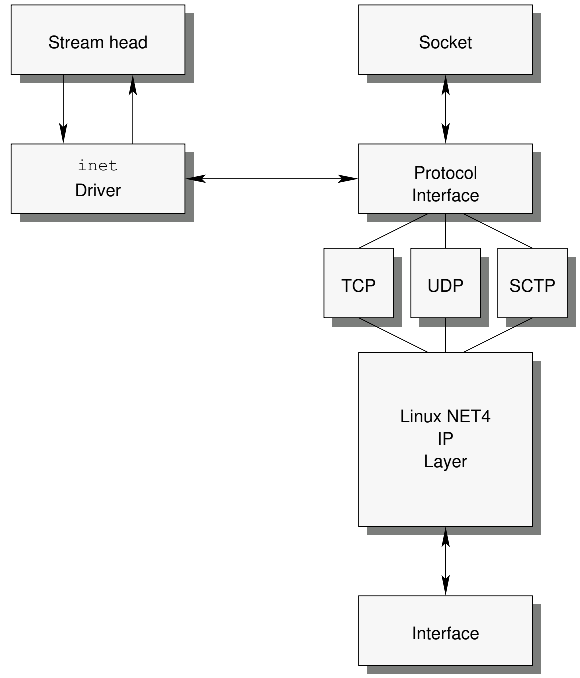

Figure 2: XTI over Sockets `inet` Driver

Read side processing. Read side processing uses standard STREAMS flow control mechanisms as are described for TPI, below. A unique characteristic of the read side processing fro the XTI over Sockets driver is that data is first copied from the internal socket to a STREAMS message block and then copied again from the STREAMS message block to user space. This constitutes two copies per byte versus one copy per byte and has a significant impact on the performance of the driver at large message sizes.[8]

Buffering. Buffering uses standard STREAMS queueing and flow control mechanisms as are described for TPI, below.

Scheduling. Scheduling resulting from queueing and flow control are the same as described for TPI below. Considering that the internal socket used by the driver is on the loop-back interface, data written on the sending socket appears immediately at the receiving socket or is discarded.

3.2.2 STREAMS TPI

The STREAMS TPI implementation of UDP is a direct STREAMS implementation that uses the `udp` driver developed by the *OpenSS7 Project* [SS7]. As illustrated in *Figure 3*, this driver interfaces to Linux at the network layer, but provides a complete STREAMS implementation of the transport layer. Interfacing with Linux at the network layer provides for de-multiplexed STREAMS architecture [RBD97]. The driver presents the Transport Provider Interface (TPI) [TPI99] for use by upper level modules and the XTI library [XTI99].

Linux Fast-STREAMS also provides a raw IP driver (`raw`) and an SCTP driver (`sctp`) that operate in the same fashion as the

7. This expectation of peformance impact is held up by the test results.
8. This expectation of peformance impact is held up by the test results.

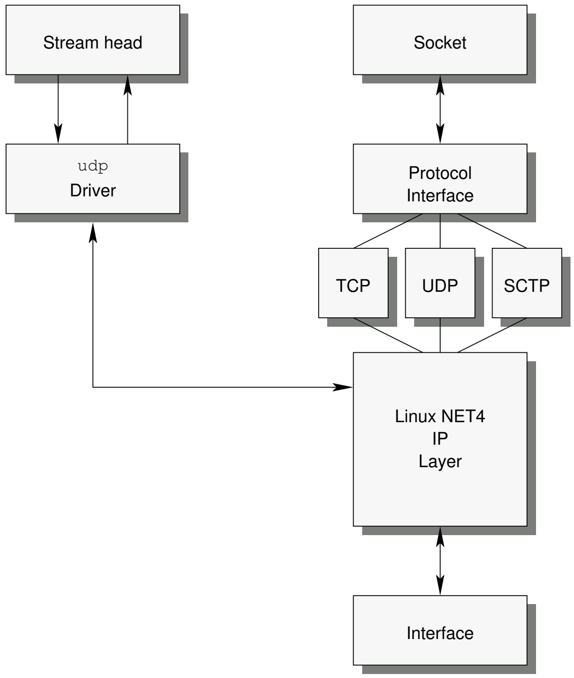

Figure 3: STREAMS `udp` Driver

`udp` driver. That is, performing all transport protocol functions within the driver and interfacing to the Linux NET4 IP layer. One of the project objectives of performing the current testing was to determine whether it would be worth the effort to write a STREAMS transport implementation of TCP, the only missing component in the TCP/IP suite that necessitates the continued support of the XTI over Sockets (`inet`) driver.

Write side processing. Write side processing follows standard STREAMS flow control. When a write occurs at the *Stream head*, the *Stream head* checks for downstream flow control on the write queue. If the *Stream* is flow controlled, the calling process is blocked or the write system call fails (`EAGAIN`). When the *Stream* is not flow controlled, user data is transferred to allocated message blocks and passed downstream. When the message blocks arrive at a downstream queue, the count of the data in the message blocks is added to to the queue count. If the queue count exceeds the high water mark defined for the queue, the queue is marked full and subsequent upstream flow control tests will fail.

Read side processing. Read side processing follows standard STREAMS flow control. When a read occurs at the *Stream head*, the *Stream head* checks the read queue for messages. If the read queue has no messages queued, the queue is marked to be enabled when messages arrive and the calling process is either blocked or the system call returns an error (`EAGAIN`). If messages exist on the read queue, they are dequeued and data copied from the message blocks to the user supplied buffer. If the message block is completely consumed, it is freed; otherwise, the message block is placed back on the read queue with the remaining data.

Buffering. Buffering follows the standard STREAMS queueing and flow control mechanisms. When a queue is found empty by a reading process, the fact that the queue requires service is recorded. Once the first message arrives at the queue following a process finding the queue empty, the queue's service procedure will be scheduled with the STREAMS scheduler. When a queue is tested for flow control and the queue is found to be full, the fact that a process wishes to write the to queue is recorded. When the count of the data on the queue falls beneath the low water mark, previous queues will be back enabled (that is, their service procedures will be scheduled with the STREAMS scheduler).

Scheduling. When a queue downstream from the *stream head* write queue is full, writing processes either block or fail with an error (`EAGAIN`). When the forward queue's count falls below its *low water mark*, the *stream head* write queue is back-enabled. Back-enabling consists of scheduling the queue's service procedure for execution by the STREAMS scheduler. Only later, when the STREAMS scheduler runs pending tasks, does any writing process blocked on flow control get woken.

When a *stream head* read queue is empty and a reading processes either block or fail with an error (*EAGAIN*). When a message arrives at the *stream head* read queue, the service procedure associated with the queue is scheduled for later execution by the STREAMS scheduler. Only later, when the STREAMS scheduler runs pending tasks, does any reading process blocked awaiting messages get awoken.

4 Method

To test the performance of STREAMS networking, the *Linux Fast-STREAMS* package was used [LfS]. The *Linux Fast-STREAMS* package builds and installs Linux loadable kernel modules and includes the modified `netperf` and `iperf` programs used for testing.

Test Program. One program used is a version of the `netperf` network performance measurement tool developed and maintained by Rick Jones for *Hewlett-Packard*. This modified version is available from the *OpenSS7 Project* [Jon07]. While the program is able to test using both POSIX Sockets and XTI STREAMS interfaces, modifications were required to the package to allow it to compile for *Linux Fast-STREAMS*.

The `netperf` program has many options. Therefore, a benchmark script (called `netperf_benchmark`) was used to obtain repeatable raw data for the various machines and distributions tested. This benchmark script is included in the `netperf` distribution available from the *OpenSS7 Project* [Jon07]. A listing of this script is provided in *Appendix A*.

4.1 Implementations Tested

The following implementations were tested:

UDP Sockets This is the Linux NET4 Sockets implementation of UDP, described in *Section* **??**, with normal scheduling priorities. Normal scheduling priority means invoking the sending and receiving processes without altering their run-time scheduling priority.

UDP Sockets with artificial process priorities.

STREAMS XTIoS UDP. This is the OpenSS7 STREAMS implementation of XTI over Sockets for UDP, described in *Section 3.2.1.* This implementation is tested using normal run-time scheduling priorities.

STREAMS TPI SCTP. This is the OpenSS7 STREAMS implementation of UDP using XTI/TPI directly, described in *Section 3.2.2.* This implementation is tested using normal run-time scheduling priorities.

4.2 Distributions Tested

To remove the dependence of test results on a particular Linux kernel or machine, various Linux distributions were used for testing. The distributions tested are as follows:

Distribution	Kernel
RedHat 7.2	2.4.20-28.7
WhiteBox 3	2.4.27
CentOS 4	2.6.9-5.0.3.EL
SuSE 10.0 OSS	2.6.13-15-default
Ubuntu 6.10	2.6.17-11-generic
Ubuntu 7.04	2.6.20-15-server
Fedora Core 6	2.6.20-1.2933.fc6

4.3 Test Machines

To remove the dependence of test results on a particular machine, various machines were used for testing as follows:

Hostname	Processor	Memory	Architecture
porky	2.57GHz PIV	1Gb (333MHz)	i686 UP
pumbah	2.57GHz PIV	1Gb (333MHz)	i686 UP
daisy	3.0GHz i630 HT	1Gb (400MHz)	x86_64 SMP
mspiggy	1.7GHz PIV	1Gb (333MHz)	i686 UP

5 Results

The results for the various distributions and machines is tabulated in *Appendix B*. The data is tabulated as follows:

Performance. Performance is charted by graphing the number of messages sent and received per second against the logarithm of the message send size.

Delay. Delay is charted by graphing the number of seconds per send and receive against the sent message size. The delay can be modelled as a fixed overhead per send or receive operation and a fixed overhead per byte sent. This model results in a linear graph with the zero x-intercept representing the fixed per-message overhead, and the slope of the line representing the per-byte cost. As all implementations use the same primary mechanism for copying bytes to and from user space, it is expected that the slope of each graph will be similar and that the intercept will reflect most implementation differences.

Throughput. Throughput is charted by graphing the logarithm of the product of the number of messages per second and the message size against the logarithm of the message size. It is expected that these graphs will exhibit strong log-log-linear (power function) characteristics. Any curvature in these graphs represents throughput saturation.

Improvement. Improvement is charted by graphing the quotient of the bytes per second of the implementation and the bytes per second of the Linux sockets implementation as a percentage against the message size. Values over 0% represent an improvement over Linux sockets, whereas values under 0% represent the lack of an improvement.

The results are organized in the sections that follow in order of the machine tested.

5.1 Porky

Porky is a 2.57GHz Pentium IV (i686) uniprocessor machine with 1Gb of memory. Linux distributions tested on this machine are as follows:

Distribution	Kernel
Fedora Core 6	2.6.20-1.2933.fc6
CentOS 4	2.6.9-5.0.3.EL
SuSE 10.0 OSS	2.6.13-15-default
Ubuntu 6.10	2.6.17-11-generic
Ubuntu 7.04	2.6.20-15-server

5.1.1 Fedora Core 6

Fedora Core 6 is the most recent full release Fedora distribution. This distribution sports a 2.6.20-1.2933.fc6 kernel with the latest patches. This is the `x86` distribution with recent updates.

Performance. *Figure 4* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

Delay. *Figure 5* plots the average message delay of UDP Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

From the figure, it can be seen that the slope of the delay graph for STREAMS and Sockets are about the same. This is expected as both implementations use the same function to copy message bytes to and from user space. The slope of the XTI over Sockets graph is over twice the slope of the Sockets graph which reflects the fact that XTI over Sockets performs multiple copies of the data: two copies on the send side and two copies on the receive side.

The intercept for STREAMS is lower than Sockets, indicating that STREAMS has a lower per-message overhead than Sockets, despite the fact that the destination address is being copied to and from user space for each message.

Throughput. *Figure 6* plots the effective throughput of UDP Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

As can be seen from the figure, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation. The slight concave downward curvature of the graphs at large message sizes indicates some degree of saturation.

Improvement. *Figure 7* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (approx. 30% improvement) at message sizes below 1024 bytes. Perhaps surprising is that the XTI over Sockets approach rivals (95%) Sockets alone at small message sizes (where multiple copies are not controlling).

The results for Fedora Core 6 on Porky are, for the most part, similar to the results from other distributions on the same host and also similar to the results for other distributions on other hosts.

5.1.2 CentOS 4.0

CentOS 4.0 is a clone of the RedHat Enterprise 4 distribution. This is the `x86` version of the distribution. The distribution sports a 2.6.9-5.0.3.EL kernel.

Performance. *Figure 8* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

As can be seen from the figure, *Linux Fast-STREAMS* outperforms Linux at all message sizes. Also, and perhaps surprisingly, the XTI over Sockets implementation also performs as well as Linux at lower message sizes.

Delay. *Figure 9* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

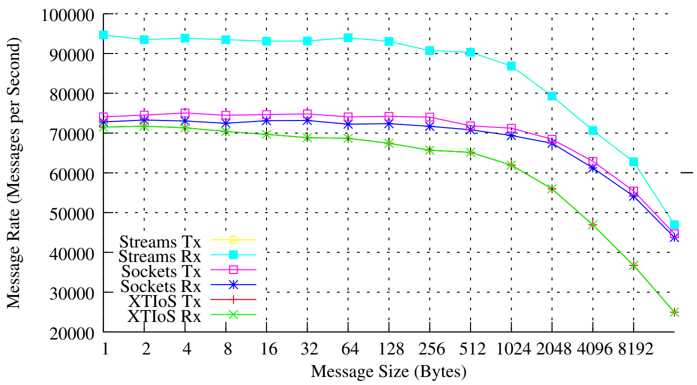

Figure 4: Fedora Core 6 on Porky Performance

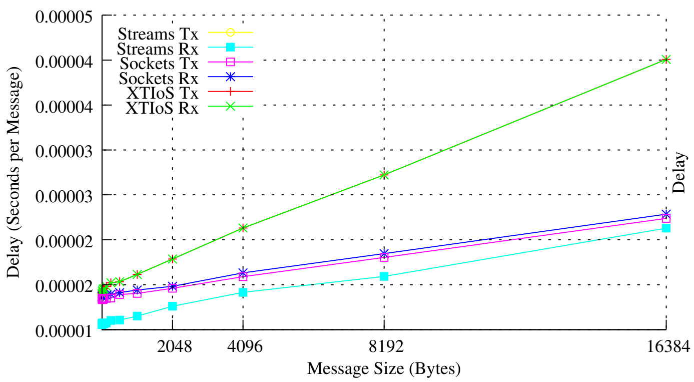

Figure 5: Fedora Core 6 on Porky Delay

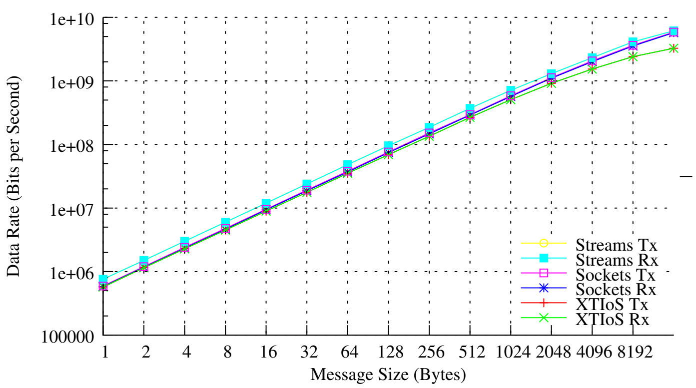

Figure 6: Fedora Core 6 on Porky Throughput

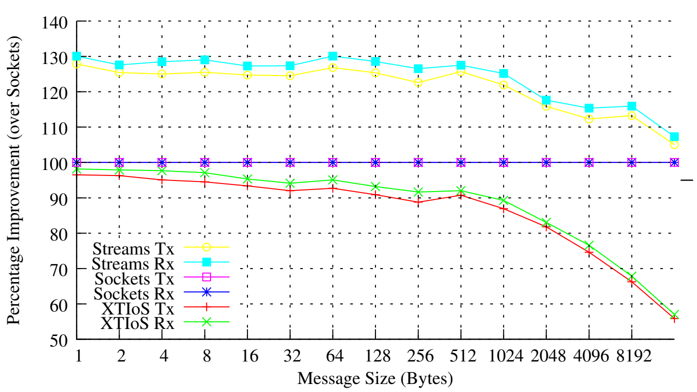

Figure 7: Fedora Core 6 on Porky Comparison

Both STREAMS and Sockets exhibit the same slope, and XTI over Sockets exhibits over twice the slope, indicating that copies of data control the per-byte characteristics. STREAMS exhibits a lower intercept than both other implementations, indicating that STREAMS has the lowest per-message overhead, regardless of copying the destination address to and from the user with each sent and received message.

Throughput. *Figure 10* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

As can be seen from the figure, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation. Again, the slight concave downward curvature at large memory sizes indicates memory bus saturation.

Improvement. *Figure 11* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (approx. 30-40% improvement) at message sizes below 1024 bytes. Perhaps surprising is that the XTI over Sockets approach rivals (97%) Sockets alone at small message sizes (where multiple copies are not controlling).

The results for CentOS on Porky are, for the most part, similar to the results from other distributions on the same host and also similar to the results for other distributions on other hosts.

5.1.3 SuSE 10.0 OSS

SuSE 10.0 OSS is the public release version of the SuSE/Novell distribution. There have been two releases subsequent to this one: the 10.1 and recent 10.2 releases. The SuSE 10 release sports a 2.6.13 kernel and the 2.6.13-15-default kernel was the tested kernel.

Performance. *Figure 12* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

Delay. *Figure 13* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

Again, STREAMS and Sockets exhibit the same slope, and XTI over Sockets more than twice the slope. STREAMS again has a significantly lower intercept and the XTI over Sockets and Sockets intercepts are similar, indicating that STREAMS has a smaller per-message overhead, despite copying destination addresses with each message.

Throughput. *Figure 14* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

As can be seen from *Figure 14*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

Improvement. *Figure 15* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (25-30%) at all message sizes.

The results for SuSE 10 OSS on Porky are, for the most part, similar to the results from other distributions on the same host and also similar to the results for other distributions on other hosts.

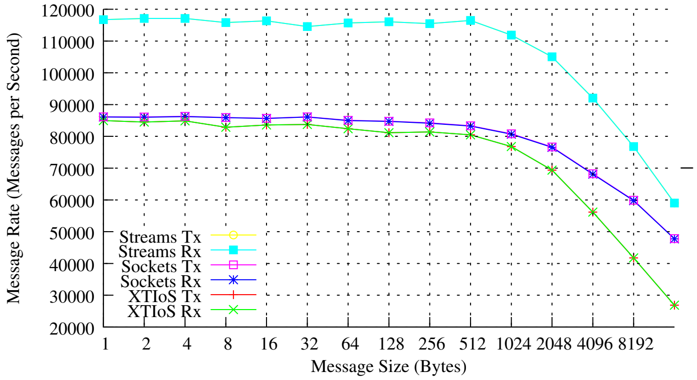

Figure 8: CentOS on Porky Performance

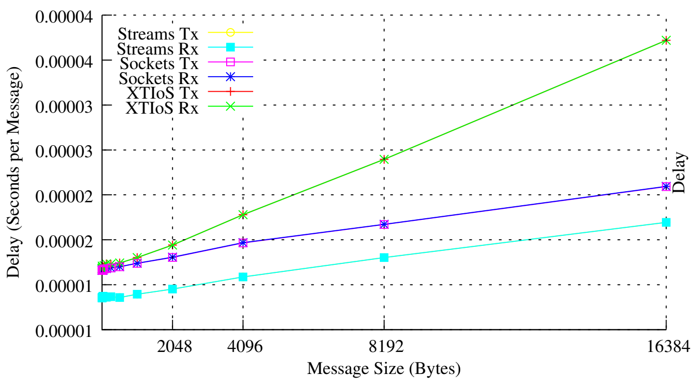

Figure 9: CentOS on Porky Delay

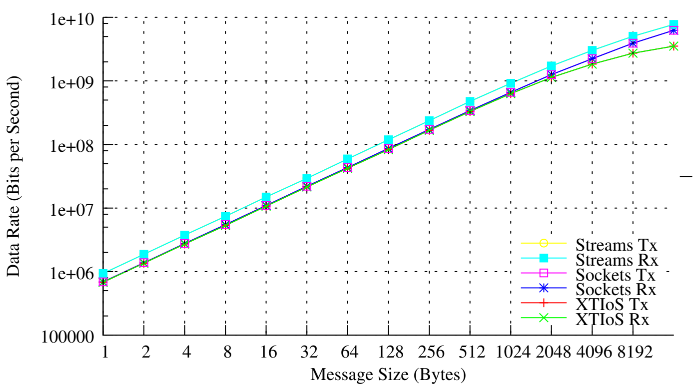

Figure 10: CentOS on Porky Throughput

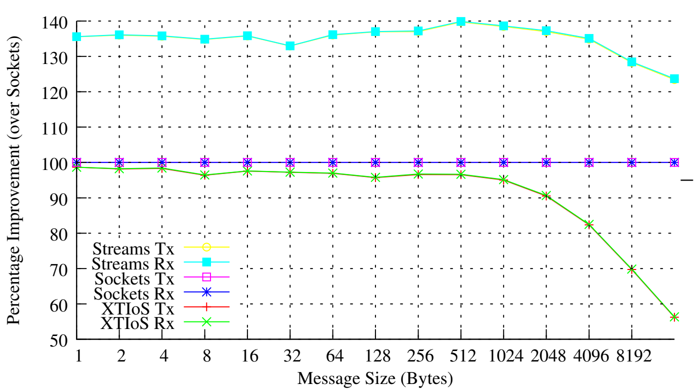

Figure 11: CentOS on Porky Comparison

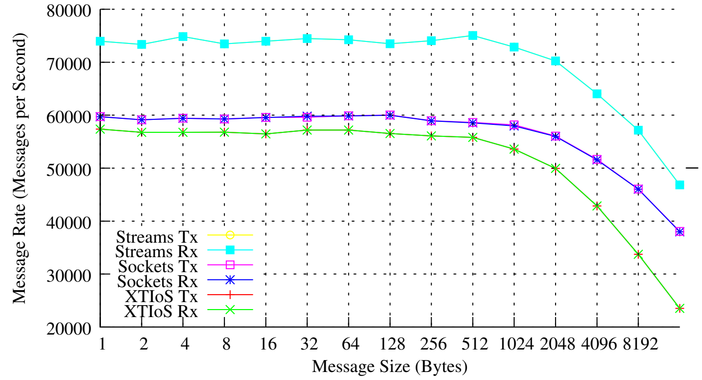

Figure 12: SuSE on Porky Performance

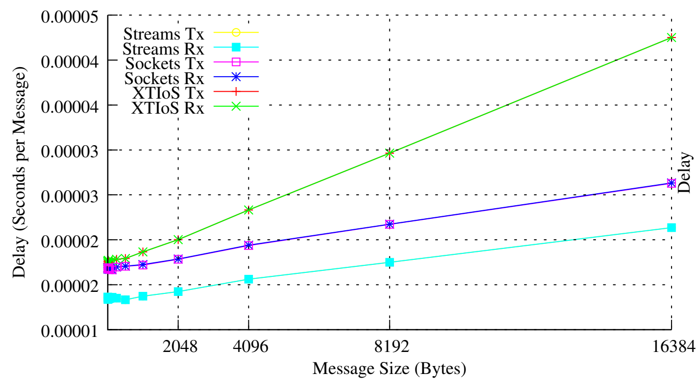

Figure 13: SuSE on Porky Delay

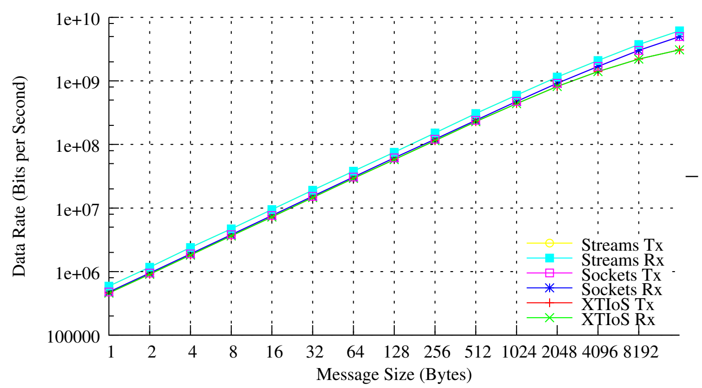

Figure 14: SuSE on Porky Throughput

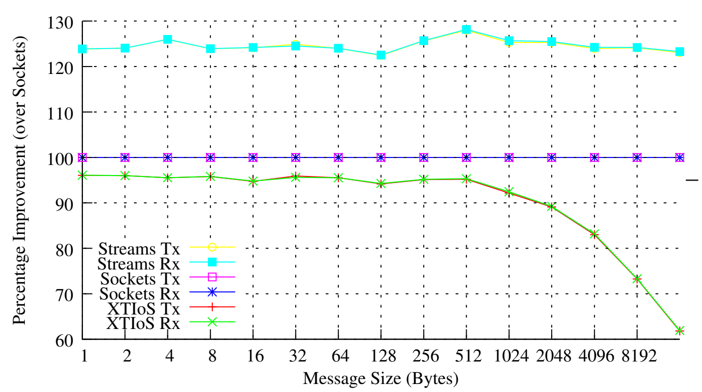

Figure 15: SuSE on Porky Comparison

5.1.4 Ubuntu 6.10

Ubuntu 6.10 is the current release of the Ubuntu distribution. The Ubuntu 6.10 release sports a 2.6.15 kernel. The tested distribution had current updates applied.

Performance. *Figure 16* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates marginal improvements (approx. 5%) at all message sizes.

Delay. *Figure 17* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates marginal improvements at all message sizes.

Although STREAMS exhibits the same slope (per-byte processing overhead) as Sockets, Ubuntu and the 2.6.17 kernel are the only combination where the STREAMS intercept is not significantly lower than Sockets. Also, the XTI over Sockets slope is steeper and the XTI over Sockets intercept is much larger than Sockets alone.

Throughput. *Figure 18* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates marginal improvements at all message sizes.

As can be seen from *Figure 18*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

Improvement. *Figure 19* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates marginal improvements (approx. 5%) at all message sizes.

Unbuntu is the only distribution tested where STREAMS does not show significant improvements over Sockets. Nevertheless, STREAMS does show marginal improvement (approx. 5%) over all message sizes and performed better than Sockets at all message sizes.

5.1.5 Ubuntu 7.04

Ubuntu 7.04 is the current release of the Ubuntu distribution. The Ubuntu 7.04 release sports a 2.6.20 kernel. The tested distribution had current updates applied.

Performance. *Figure 20* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (approx. 20-60%) at all message sizes.

Delay. *Figure 21* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

STREAMS and Sockets exhibit the slope, and XTI over Sockets more than twice the slope. STREAMS, however, has a significantly lower intercept and XTI over Sockets and Sockets intercepts are similar, indicating that STREAMS has a smaller per-message overhead, despite copying destination addresses with each message.

Throughput. *Figure 22* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

As can be seen from *Figure 22*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

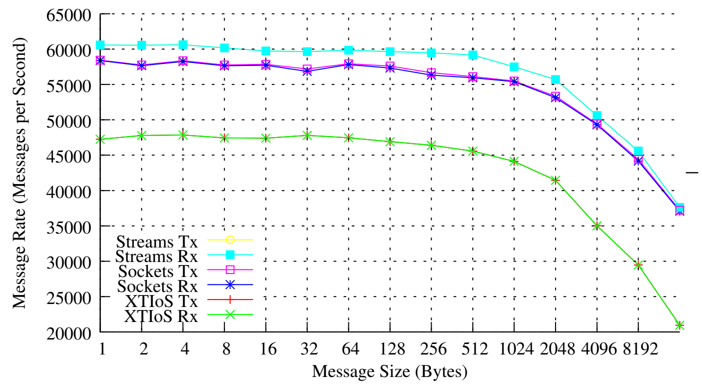

Figure 16: Ubuntu 6.10 on Porky Performance

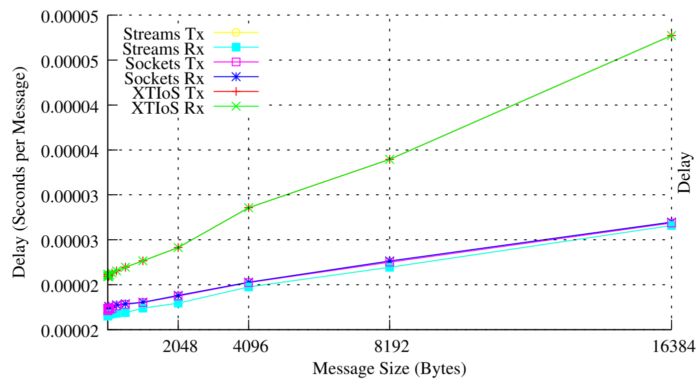

Figure 17: Ubuntu 6.10 on Porky Delay

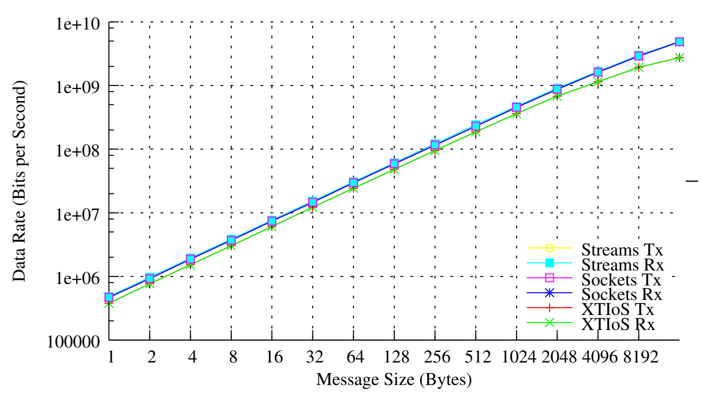

Figure 18: Ubuntu 6.10 on Porky Throughput

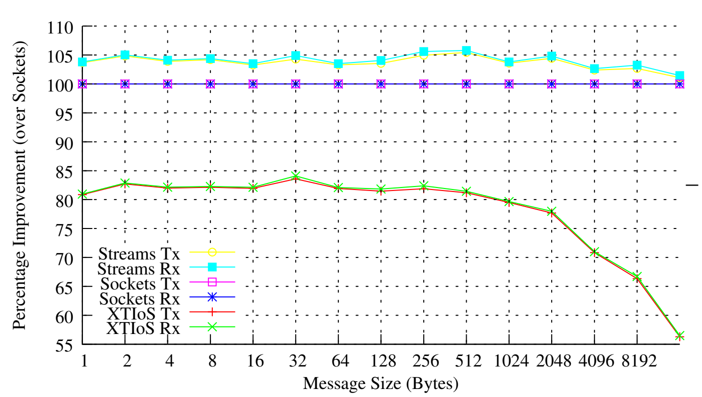

Figure 19: Ubuntu 6.10 on Porky Comparison

Improvement. *Figure 23* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (approx. 20-60%) at all message sizes.

The results for Ubuntu 7.04 on Porky are, for the most part, similar to the results from other distributions on the same host and also similar to the results for other distributions on other hosts.

5.2 Pumbah

Pumbah is a 2.57GHz Pentium IV (i686) uniprocessor machine with 1Gb of memory. This machine differs from Porky in memory type only (Pumbah has somewhat faster memory than Porky.) Linux distributions tested on this machine are as follows:

Distribution	Kernel
RedHat 7.2	2.4.20-28.7

Pumbah is a control machine and is used to rule out differences between recent 2.6 kernels and one of the oldest and most stable 2.4 kernels.

5.2.1 RedHat 7.2

RedHat 7.2 is one of the oldest (and arguably the most stable) glibc2 based releases of the RedHat distribution. This distribution sports a 2.4.20-28.7 kernel. The distribution has all available updates applied.

Performance. *Figure 24* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes, and staggering improvements at large message sizes.

Delay. *Figure 25* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes, and staggering improvements at large message sizes.

The slope of the STREAMS delay curve is much lower than (almost half that of) the Sockets delay curve, indicating that STREAMS is exploiting some memory efficiencies not possible in the Sockets implementation.

Throughput. *Figure 26* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates improvements at all message sizes.

As can be seen from *Figure 26*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

The Linux NET4 UDP implementation results deviate more sharply from power function behaviour at high message sizes. This also, is rather different that the 2.6 kernel situation. One contributing factor is the fact that neither the send nor receive buffers can be set above 65,536 bytes on this version of Linux 2.4 kernel. Tests were performed with send and receive buffer size requests of 131,072 bytes. Both the STREAMS XTI over Sockets UDP implementation and the Linux NET4 UDP implementation suffer from the maximum buffer size, whereas, the STREAMS UDP implementation implements and permits the larger buffers.

Improvement. *Figure 27* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements all message sizes.

The more dramatic improvements over Linux NET4 UDP and XTI over Sockets UDP is likely due in part to the restriction on buffer sizes in 2.4 as described above.

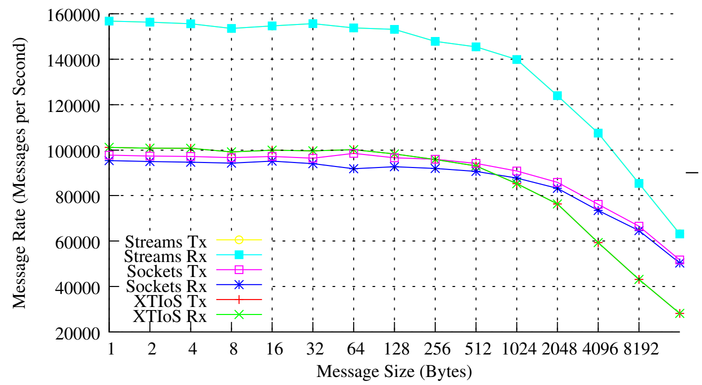

Figure 20: Ubuntu 7.04 on Porky Performance

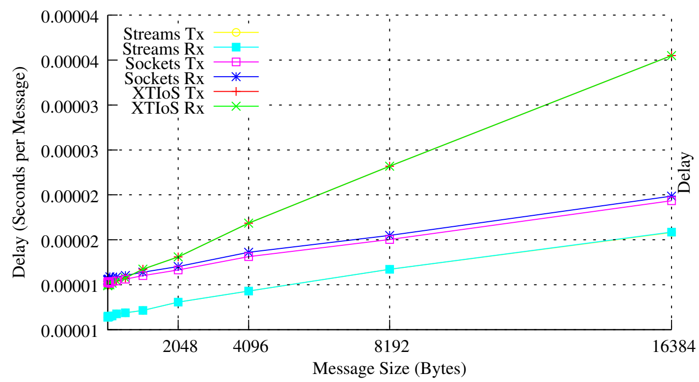

Figure 21: Ubuntu 7.04 on Porky Delay

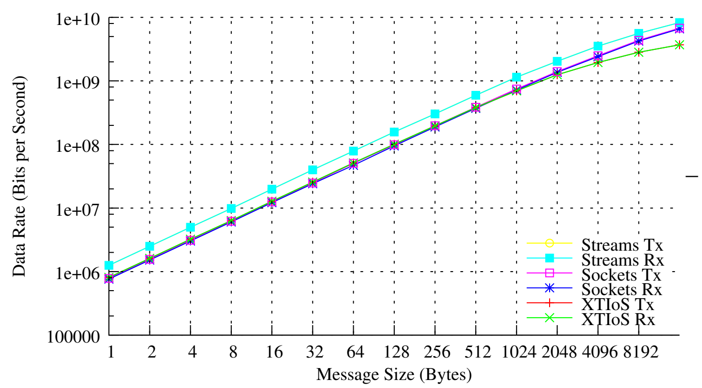

Figure 22: Ubuntu 7.04 on Porky Throughput

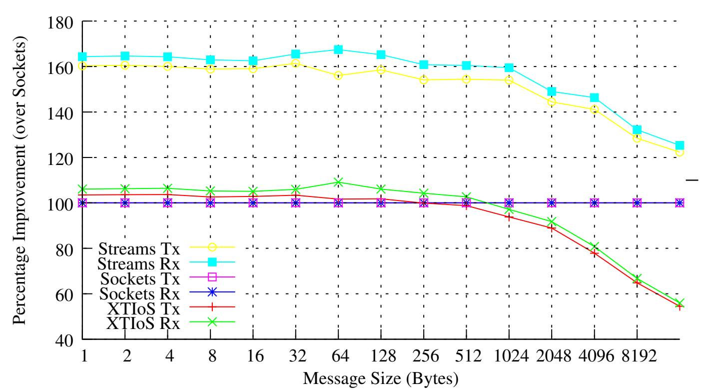

Figure 23: Ubuntu 7.04 on Porky Comparison

Unfortunately, the RedHat 7.2 system does not appear to have acted as a very good control system. The differences in maximum buffer size make any differences from other tested behaviour obvious.

5.3 Daisy

Daisy is a 3.0GHz i630 (x86_64) hyper-threaded machine with 1Gb of memory. Linux distributions tested on this machine are as follows:

Distribution	Kernel
Fedora Core 6	2.6.20-1.2933.fc6
CentOS 5.0	2.6.18-8.1.3.el5

This machine is used as an SMP control machine. Most of the tests were performed on uniprocessor non-hyper-threaded machines. This machine is hyper-threaded and runs full SMP kernels. This machine also supports EMT64 and runs `x86_64` kernels. It is used to rule out both SMP differences as well as 64-bit architecture differences.

5.3.1 Fedora Core 6 (x86_64)

Fedora Core 6 is the most recent full release Fedora distribution. This distribution sports a 2.6.20-1.2933.fc6 kernel with the latest patches. This is the `x86_64` distribution with recent updates.

Performance. *Figure 28* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

Delay. *Figure 29* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

The slope of the delay curve either indicates that Sockets is using slightly larger buffers than STREAMS, or that Sockets is somehow exploiting some per-byte efficiencies at larger message sizes not achieved by STREAMS. Nevertheless, the STREAMS intercept is so low that the delay curve for STREAMS is everywhere beneath that of Sockets.

Throughput. *Figure 30* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

As can be seen from *Figure 30*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

Improvement. *Figure 31* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (approx. 40% improvement) at message sizes below 1024 bytes. That STREAMS UDP gives a 40% improvement over a wide range of message sizes on SMP is a dramatic improvement. Statements regarding STREAMS networking running poorer on SMP than on UP are quite wrong, at least with regard to *Linux Fast-STREAMS*.

5.3.2 CentOS 5.0 (x86_64)

CentOS 5.0 is the most recent full release CentOS distribution. This distribution sports a 2.6.18-8.1.3.el5 kernel with the latest patches. This is the `x86_64` distribution with recent updates.

Performance. *Figure 32* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

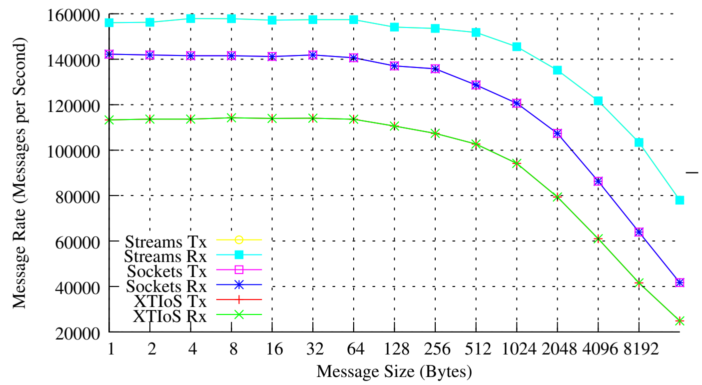

Figure 24: RedHat 7.2 on Pumbah Performance

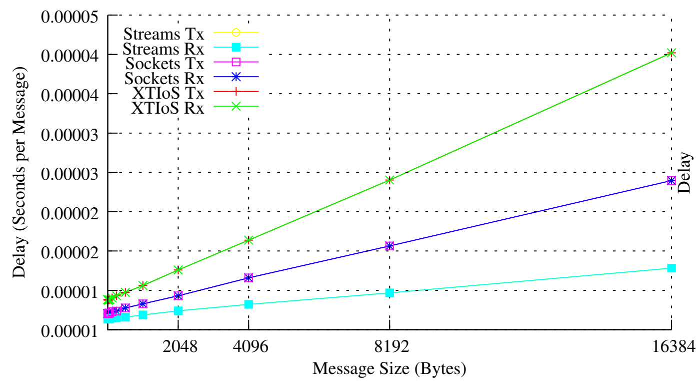

Figure 25: RedHat 7.2 on Pumbah Delay

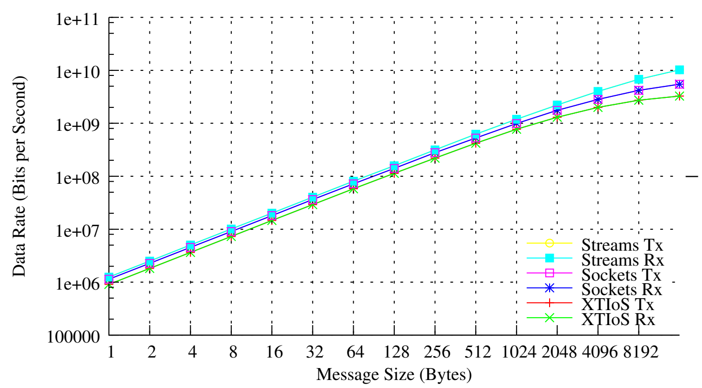

Figure 26: RedHat 7.2 on Pumbah Throughput

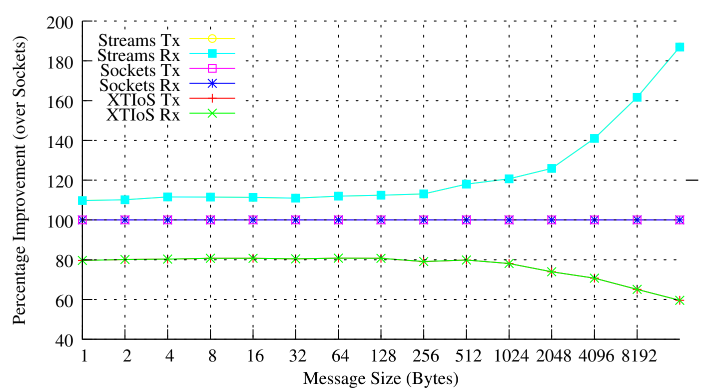

Figure 27: RedHat 7.2 on Pumbah Comparison

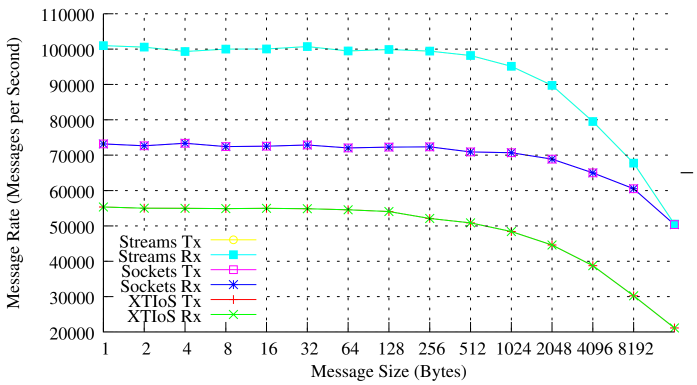

Figure 28: Fedora Core 6 on Daisy Performance

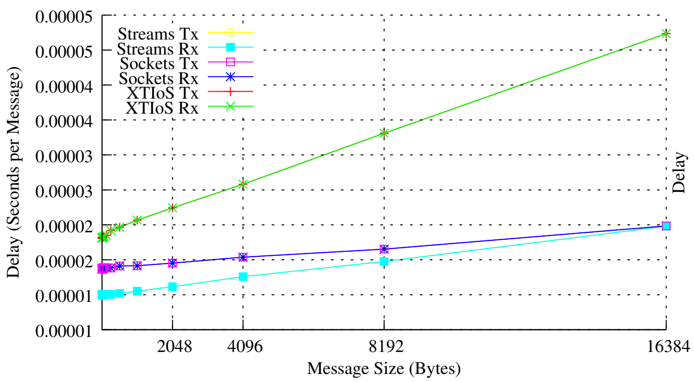

Figure 29: Fedora Core 6 on Daisy Delay

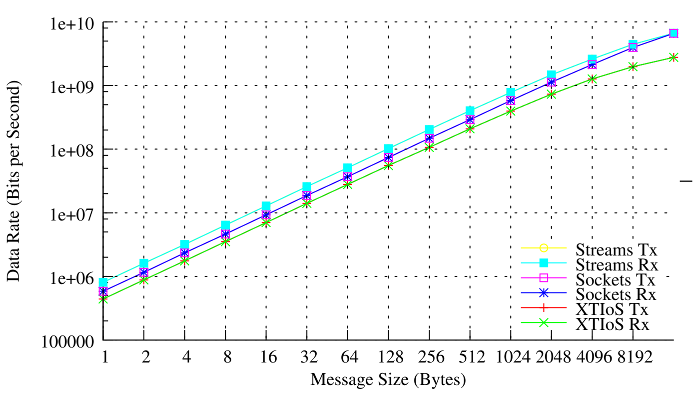

Figure 30: Fedora Core 6 on Daisy Throughput

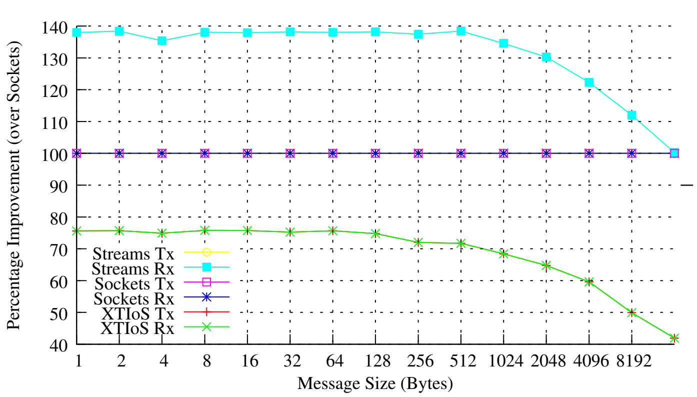

Figure 31: Fedora Core 6 on Daisy Comparison

Delay. *Figure 33* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes.

The slope of the delay curve either indicates that Sockets is using slightly larger buffers than STREAMS, or that Sockets is somehow exploiting some per-byte efficiencies at larger message sizes not achieved by STREAMS. Nevertheless, the STREAMS intercept is so low that the delay curve for STREAMS is everywhere beneath that of Sockets.

Throughput. *Figure 34* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes.

As can be seen from *Figure 34*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

Improvement. *Figure 35* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements (approx. 40% improvement) at message sizes below 1024 bytes. That STREAMS UDP gives a 40% improvement over a wide range of message sizes on SMP is a dramatic improvement. Statements regarding STREAMS networking running poorer on SMP than on UP are quite wrong, at least with regard to *Linux Fast-STREAMS*.

5.4 Mspiggy

Mspiggy is a 1.7Ghz Pentium IV (M-processor) uniprocessor notebook (Toshiba Satellite 5100) with 1Gb of memory. Linux distributions tested on this machine are as follows:

Distribution	Kernel
SuSE 10.0 OSS	2.6.13-15-default

Note that this is the same distribution that was also tested on Porky. The purpose of testing on this notebook is to rule out the differences between machine architectures on the test results. Tests performed on this machine are control tests.

5.4.1 SuSE 10.0 OSS

SuSE 10.0 OSS is the public release version of the SuSE/Novell distribution. There have been two releases subsequent to this one: the 10.1 and recent 10.2 releases. The SuSE 10 release sports a 2.6.13 kernel and the 2.6.13-15-default kernel was the tested kernel.

Performance. *Figure 36* plots the measured performance of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes, and staggering improvements at large message sizes.

Delay. *Figure 37* plots the average message delay of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements at all message sizes, and staggering improvements at large message sizes.

The slope of the STREAMS delay curve is much lower than (almost half that of) the Sockets delay curve, indicating that STREAMS is exploiting some memory efficiencies not possible in the Sockets implementation.

Throughput. *Figure 38* plots the effective throughput of Sockets compared to XTI over Socket and XTI approaches. STREAMS demonstrates improvements at all message sizes.

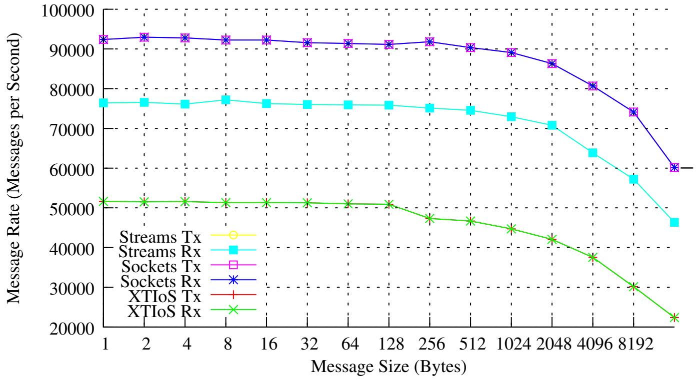

Figure 32: CentOS 5.0 on Daisy Performance

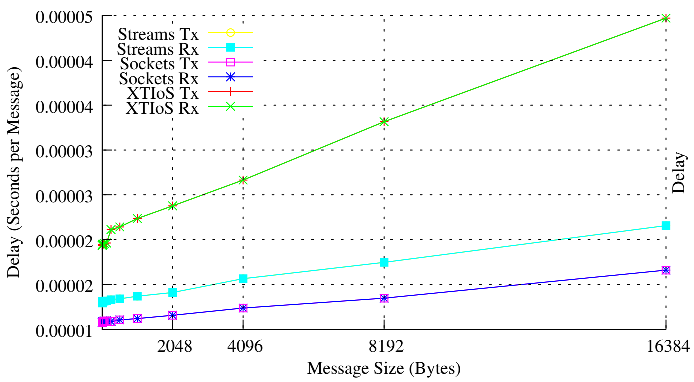

Figure 33: CentOS 5.0 on Daisy Delay

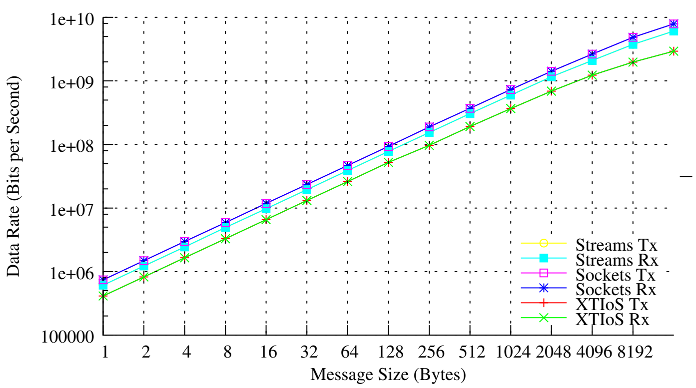

Figure 34: CentOS 5.0 on Daisy Throughput

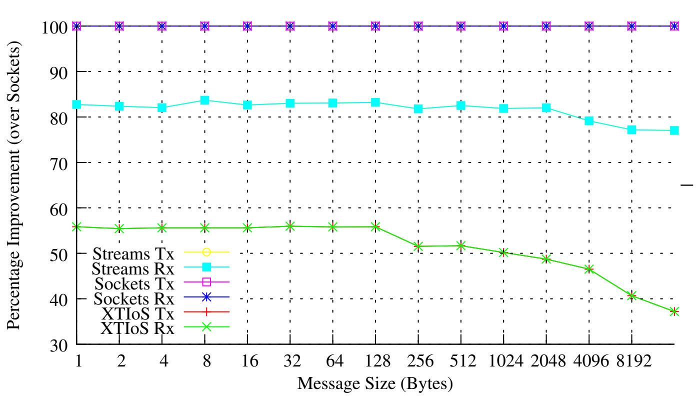

Figure 35: CentOS 5.0 on Daisy Comparison

As can be seen from *Figure 38*, all implementations exhibit strong power function characteristics (at least at lower write sizes), indicating structure and robustness for each implementation.

The Linux NET4 UDP implementation results deviate more sharply from power function behaviour at high message sizes. One contributing factor is the fact that neither the send nor receive buffers can be set above about 111,000 bytes on this version of Linux 2.6 kernel running on this speed of processor. Tests were performed with send and receive buffer size requests of 131,072 bytes. Both the STREAMS XTI over Sockets UDP implementation and the Linux NET4 UDP implementation suffer from the maximum buffer size, whereas, the STREAMS UDP implementation implements and permits the larger buffers.

Improvement. *Figure 39* plots the comparison of Sockets to XTI over Socket and XTI approaches. STREAMS demonstrates significant improvements all message sizes.

The more dramatic improvements over Linux NET4 UDP and XTI over Sockets UDP is likely due in part to the restriction on buffer sizes in 2.6 on slower processors as described above.

Unfortunately, this SuSE 10.0 OSS system does not appear to have acted as a very good control system. The differences in maximum buffer size make any differences from other tested behaviour obvious.

6 Analysis

With some caveats as described at the end of this section, the results are consistent enough across the various distributions and machines tested to draw some conclusions regarding the efficiency of the implementations tested. This section is responsible for providing an analysis of the results and drawing conclusions consistent with the experimental results.

6.1 Discussion

The test results reveal that the maximum throughput performance, as tested by the `netperf` program, of the STREAMS implementation of UDP is superior to that of the Linux NET4 Sockets implementation of UDP. In fact, STREAMS implementation performance at smaller message sizes is significantly (as much as 30-40%) greater than that of Linux NET4 UDP. As the common belief is that STREAMS would exhibit poorer performance, this is perhaps a startling result to some.

Looking at both implementations, the differences can be described by implementation similarities and differences:

Send processing. When Linux NET4 UDP receives a send request, the available send buffer space is checked. If the current data would cause the send buffer fill to exceed the send buffer maximum, either the calling process blocks awaiting available buffer, or the system call returns with an error (`ENOBUFS`). If the current send request will fit into the send buffer, a socket buffer (`skbuff`) is allocated, data is copied from user space to the buffer, and the socket buffer is dispatched to the IP layer for transmission.

Linux 2.6 kernels have an amazing amount of special case code that gets executed for even a simple UDP send operation. Linux 2.4 kernels are far more direct. The result is the same, even though they are different in the depths to which they must delve before discovering that a send is just a simple send. This might explain part of the rather striking differences between the performance comparison between STREAMS and NET4 on 2.6 and 2.4 kernels.

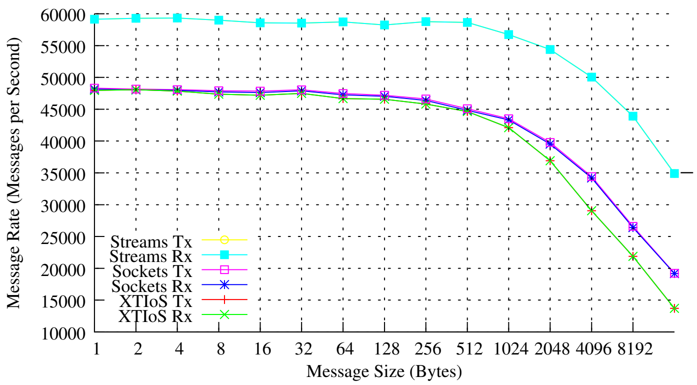

Figure 36: SuSE 10.0 OSS Mspiggy Performance

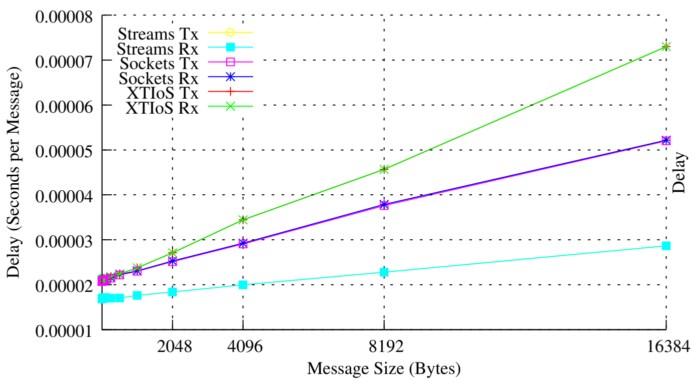

Figure 37: SuSE 10.0 OSS Mspiggy Delay

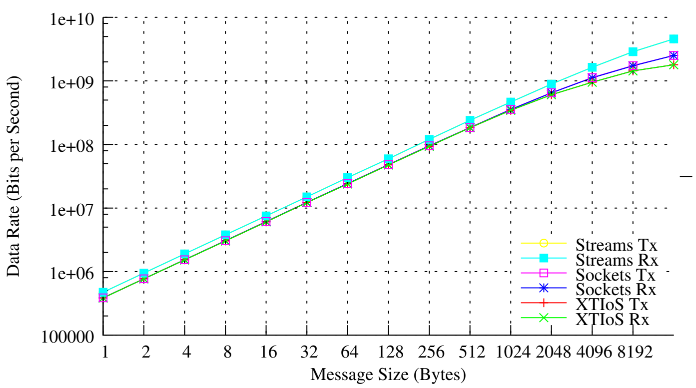

Figure 38: SuSE 10.0 OSS Mspiggy Throughput

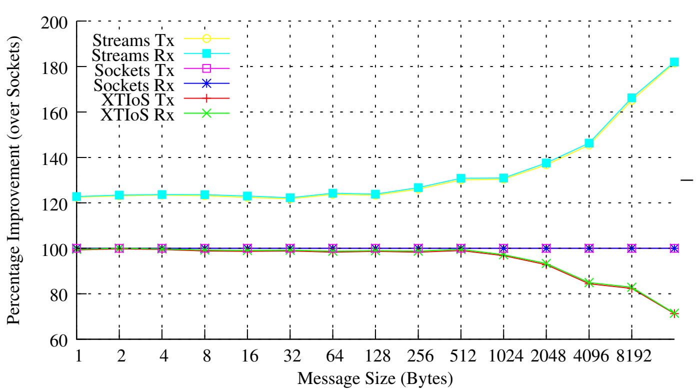

Figure 39: SuSE 10.0 OSS Mspiggy Comparison

When the STREAMS Stream head receives a putmsg(2) request, it checks downstream flow control. If the Stream is flow controlled downstream, either the calling process blocks awaiting succession of flow control, or the putmsg(2) system call returns with an error (`EAGAIN`). if the Stream is not flow controlled on the write side, message blocks are allocated to hold the control and data portions of the request and the message blocks are passed downstream to the driver. When the driver receives an `M_DATA` or `M_PROTO` message block from the Stream head, in its put procedure, it simply queues it to the driver write queue with putq(9). putq(9) will result in the enabling of the service procedure for the driver write queue under the proper circumstances. When the service procedure runs, messages will be dequeued from the driver write queue transformed into IP datagrams and sent to the IP layer for transmission on the network interface.

Linux Fast-STREAMS has a feature whereby the driver can request that the Stream head allocate a Linux socket buffer (`skbuff`) to hold the data buffer associated with an allocated message block. The STREAMS UDP driver utilizes this feature (but the STREAMS XTIoS UDP driver cannot). STREAMS also has the feature that a write offset can be applied to all data block allocated and passed downstream. The STREAMS UDP driver uses this capability also. The write offset set by the tested driver was a maximum hard header length.

Network processing. Network processing (that is the bottom end under the transport protocol) for both implementations is effectively the same, with only minor differences. In the STREAMS UDP implementation, no `sock` structure exists, so issuing socket buffers to the network layer is performed in a slightly more direct fashion.

Loop-back processing is identical as this is performed by the Linux NET4 IP layer in both cases.

For Linux Sockets UDP, when the IP layer frees or orphans the socket buffer, the amount of data associated with the socket buffer is subtracted from the current send buffer fill. If the current buffer fill is less than 1/2 of the maximum, all processes blocked on write or blocked on poll are are woken.

For STREAMS UDP, when the IP layer frees or orphans the socket buffer, the amount of data associated with the socket buffer is subtracted from the current send buffer fill. If the current send buffer fill is less than the send buffer low water mark (`SO_SNDLOWAT` or `XTI_SNDLOWAT`), and the write queue is blocked on flow control, the write queue is enabled.

One disadvantage that it is expected would slow STREAMS UDP performance is the fact that on the sending side, a STREAMS buffer is allocated along with a `skbuff` and the `skbuff` is passed to Linux NET4 IP and the loop-back device. For Linux Sockets UDP, the same `skbuff` is reused on both sides of the interface. For STREAMS UDP, there is (currently) no mechanism for passing through the original STREAMS message block and a new message block must be allocated. This results in two message block allocations per `skbuff`.

Receive processing. Under Linux Sockets UDP, when a socket buffer is received from the network layer, a check is performed whether the associated socket is locked by a user process or not. If the associated socket is locked, the socket buffer is placed on a backlog queue awaiting later processing by the user process when it goes to release the lock. A maximum number of socket buffers are permitted to be queued against the backlog queue per socket (approx. 300).

If the socket is not locked, or if the user process is processing a backlog before releasing the lock, the message is processed: the receive socket buffer is checked and if the received message would cause the buffer to exceed its maximum size, the message is discarded and the socket buffer freed. If the received message fits into the buffer, its size is added to the current send buffer fill and the message is queued on the socket receive queue. If a process is sleeping on read or in poll, an immediate wakeup is generated.

In the STREAMS UDP implementation on the receive side, again there is no `sock` structure, so the socket locking and backlog techniques performed by UDP at the lower layer do not apply. When the STREAMS UDP implementation receives a socket buffer from the network layer, it tests the receive side of the Stream for flow control and, when not flow controlled, allocates a STREAMS buffer using esballoc(9) and passes the buffer directly to the upstream queue using putnext(9). When flow control is in effect and the read queue of the driver is not full, a STREAMS message block is still allocated and placed on the driver read queue. When the driver read queue is full, the received socket buffer is freed and the contents discarded. While different in mechanism from the socket buffer and backlog approach taken by Linux Sockets UDP, this bottom end receive mechanism is similar in both complexity and behaviour.

Buffering. For Linux Sockets, when a send side socket buffer is allocated, the true size of the socket buffer is added to the current send buffer fill. After the socket buffer has been passed to the IP layer, and the IP layer consumes (frees or orphans) the socket buffer, the true size of the socket buffer is subtracted from the current send buffer fill. When the resulting fill is less than 1/2 the send buffer maximum, sending processes blocked on send or poll are woken up. When a send will not fit within the maximum send buffer size considering the size of the transmission and the current send buffer fill, the calling process blocks or is returned an error (`ENOBUFS`). Processes that are blocked or subsequently block on poll(2) will not be woken up until the send buffer fill drops beneath 1/2 of the maximum; however, any process that subsequently attempts to send and has data that will fit in the buffer will be permitted to proceed.

STREAMS networking, on the other hand, performs queueing, flow control and scheduling on both the sender and the receiver. Sent messages are queued before delivery to the IP subsystem. Received messages from the IP subsystem are queued before delivery to the receiver. Both side implement full hysteresis high and low water marks. Queues are deemed full when they reach the *high water mark* and do not enable feeding processes or subsystems until the queue subsides to the *low water mark.*

Scheduling. Linux Sockets schedule by waking a receiving process whenever data is available in the receive buffer to be read, and waking a sending process whenever there is one-half of the send buffer available to be written. While accomplishing buffering on the receive side, full hysteresis flow control is only performed on the sending side. Due to the way that Linux handles the loop-back interface, the full hysteresis flow control on the sending side is defeated.

STREAMS networking, on the other hand, uses the queueing, flow control and scheduling mechanism of STREAMS. When messages are delivered from the IP layer to the receiving *stream head* and a receiving process is sleeping, the service procedure for the reading *stream head*'s read queue is scheduled for later execution. When the STREAMS scheduler later runs, the receiving process is awoken. When messages are sent on the sending side they are queued in the driver's write queue and the service procedure for the driver's write queue is scheduled for later execution. When the STREAMS scheduler later runs, the messages are delivered to the IP layer. When sending processes are blocked on a full driver write queue, and the count drops to the *low water mark* defined for the queue, the service procedure of the sending *stream head* is scheduled for later execution. When the STREAMS scheduler later runs, the sending process is awoken.

Linux Fast-STREAMS is designed to run tasks queued to the STREAMS scheduler on the same processor as the queueing processor or task. This avoid unnecessary context switches.

The STREAMS networking approach results in fewer blocking and wakeup events being generated on both the sending and receiving side. Because there are fewer blocking and wakeup events, there are fewer context switches. The receiving process is permitted to consume more messages before the sending process is awoken; and the sending process is permitted to generate more messages before the reading process is awoken.

Result The result of the differences between the Linux NET and the STREAMS approach is that better flow control is being exerted on the sending side because of intermediate queueing toward the IP layer. This intermediate queueing on the sending side, not present in BSD-style networking, is in fact responsible for reducing the number of blocking and wakeup events on the sender, and permits the sender, when running, to send more messages in a quantum.

On the receiving side, the STREAMS queueing, flow control and scheduling mechanisms are similar to the BSD-style software interrupt approach. However, Linux does not use software interrupts on loop-back (messages are passed directly to the socket with possible backlogging due to locking). The STREAMS approach is more sophisticated as it performs backlogging, queueing and flow control simultaneously on the read side (at the *stream head*).

6.2 Caveats

The following limitations in the test results and analysis must be considered:

6.2.1 Loop-back Interface

Tests compare performance on loop-back interface only. Several characteristics of the loop-back interface make it somewhat different from regular network interfaces:

1. Loop-back interfaces do not require checksums.
2. Loop-back interfaces have a null hard header.

   This means that there is less difference between putting each data chunk in a single packet versus putting multiple data chunks in a packet.
3. Loop-back interfaces have negligible queueing and emission times, making RTT times negligible.
4. Loop-back interfaces do not normally drop packets.
5. Loop-back interfaces preserve the socket buffer from sending to receiving interface.

   This also provides an advantage to Sockets TCP. Because STREAMS SCTP cannot pass a message block along with the socket buffer (socket buffers are orphaned before passing to the loop-back interface), a message block must also be allocated on the receiving side.

7 Conclusions

These experiments have shown that the *Linux Fast-STREAMS* implementation of STREAMS UDP as well as STREAMS UDP using XTIoS networking outperforms the Linux Sockets UDP implementation by a significant amount (up to 40% improvement).

> *The Linux Fast-STREAMS implementation of STREAMS UDP networking is superior by a significant factor across all systems and kernels tested.*

All of the conventional wisdom with regard to STREAMS and STREAMS networking is undermined by these test results for *Linux Fast-STREAMS.*

- *STREAMS is fast.*

  Contrary to the preconception that STREAMS must be slower because it is more general purpose, in fact the reverse has been shown to be true in these experiments for *Linux Fast-STREAMS*. The STREAMS flow control and scheduling mechanisms serve to adapt well and increase both code and data cache as well as scheduler efficiency.

- *STREAMS is more flexible* and *more efficient.*

  Contrary to the preconception that STREAMS trades flexibility or general purpose architecture for efficiency (that is, that STREAMS is somehow less efficient because it is more flexible and general purpose), in fact has shown to be untrue. *Linux Fast-STREAMS* is *both* more flexible *and* more efficient. Indeed, the performance gains achieved by STREAMS appear to derive from its more sophisticated queueing, scheduling and flow control model.

- *STREAMS better exploits parallelisms on SMP better than other approaches.*

  Contrary to the preconception that STREAMS must be slower due to complex locking and synchronization mechanisms, *Linux Fast-STREAMS* performed better on SMP (hyperthreaded) machines than on UP machines and outperformed Linux Sockets UDP by and even more significant factor (about 40% improvement at most message sizes). Indeed, STREAMS appears to be able to exploit inherent parallelisms that Linux Sockets is unable to exploit.

- *STREAMS networking is fast.*

  Contrary to the preconception that STREAMS networking must be slower because STREAMS is more general purpose and has a rich set of features, the reverse has been shown in these experiments for *Linux Fast-STREAMS*. By utilizing STREAMS queueing, flow control and scheduling, STREAMS UDP indeed performs better than Linux Sockets UDP.

- *STREAMS networking is neither unnecessarily complex nor cumbersome.*

  Contrary to the preconception that STREAMS networking must be poorer because of use of a complex yet general purpose framework has shown to be untrue in these experiments for *Linux Fast-STREAMS*. Also, the fact that STREAMS and Linux conform to the same standard (POSIX), means that they are no more cumbersome from a programming perspective. Indeed a POSIX conforming application will not known the difference between the implementation (with the exception that superior performance will be experienced on STREAMS networking).

8 Future Work

Local Transport Loop-back

UNIX domain sockets are the advocated primary interprocess communications mechanism in the 4.4BSD system: 4.4BSD even implements pipes using UNIX domain sockets [MBKQ97]. Linux also implements UNIX domain sockets, but uses the 4.1BSD/SVR3 legacy approach to pipes. XTI has an equivalent to the UNIX domain socket. This consists of connectionless, connection oriented, and connection oriented with orderly release loop-back transport providers. The `netperf` program has the ability to test UNIX domain sockets, but does not currently have the ability to test the XTI equivalents.

BSD claims that in 4.4BSD pipes were implemented using sockets (UNIX domain sockets) instead of using the file system as they were in 4.1BSD [MBKQ97]. One of the reasons cited for implementing pipes on Sockets and UNIX domain sockets using the networking subsystems was performance. Another paper released by the *OpenSS7 Project* [SS7] shows that experimental results on Linux file-system based pipes (using the SVR3 or 4.1BSD approaches) perform poorly when compared to STREAMS-based pipes. Because Linux uses a similar approach to file-system based pipes in implementation of UNIX domain sockets, it can be expected that UNIX domain sockets under Linux will also perform poorly when compared to loop-back transport providers under STREAMS.

Sockets interface to STREAMS

There are several mechanisms to providing BSD/POSIX Sockets interfaces to STREAMS networking [VS90] [Mar01]. The experiments in this report indicate that it could be worthwhile to complete one of these implementations for *Linux Fast-STREAMS* [Soc] and test whether STREAMS networking using the Sockets interface is also superior to Linux Sockets, just as it has been shown to be with the XTI/TPI interface.

9 Related Work

A separate paper comparing the STREAMS-based pipe implementation of *Linux Fast-STREAMS* to the legacy 4.1BSD/SVR3-style Linux pipe implementation has also been prepared. That paper also shows significant performance improvements for STREAMS attributable to similar causes.

A separate paper comparing a STREAMS-based SCTP implementation of *Linux Fast-STREAMS* to the Linux NET4 Sockets approach has also been prepared. That paper also shows significant performance improvements for STREAMS attributable to similar causes.

References

[GC94] Berny Goodheart and James Cox. *The magic garden explained: the internals of UNIX System V Release 4, an open systems design / Berny Goodheart & James Cox.* Prentice Hall, Australia, 1994. ISBN 0-13-098138-9.

[Jon07] Rick Jones. Network performance with netperf – An OpenSS7 Modified Version. http://www.openss7.org/download.html, 2007.

[LfS] Linux Fast-STREAMS – A High-Performance SVR 4.2 MP STREAMS Implementation for Linux. http://www.openss7.org/download.html.

[LiS] Linux STREAMS (LiS). http://www.openss7.org/download.html.

[LML] Linux Kernel Mailing List – Frequently Asked Questions. http://www.kernel.org/pub/linux/docs/lkml/#s9-9.

[Mar01] Jim Mario. Solaris sockets, past and present. *Unix Insider*, September 2001.

[MBKQ97] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The design and implementation of the 4.4BSD operating system.* Addison-Wesley, third edition, November 1997. ISBN 0-201-54979-4.

[OG] The Open Group. http://www.opengroup.org/.

[RBD97] Vincent Roca, Torsten Braun, and Christophe Diot. Demultiplexed architectures: A solution for efficient STREAMS-based communications stacks. *IEEE Network*, July/August 1997.

[Rit84] Dennis M. Ritchie. A Stream Input-output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984. Part 2.

[Soc] Sockets for linux fast-streams. http://www.openss7.-org/download.html.

[SS7] The OpenSS7 Project. http://www.openss7.org/.

[SUS95] Single UNIX Specification, Version 1. Open Group Publication, The Open Group, 1995. http://www.-opengroup.org/onlinepubs/.

[SUS98] Single UNIX Specification, Version 2. Open Group Publication, The Open Group, 1998. http://www.-opengroup.org/onlinepubs/.

[SUS03] Single UNIX Specification, Version 3. Open Group Publication, The Open Group, 2003. http://www.-opengroup.org/onlinepubs/.

[TLI92] Transport Provider Interface Specification, Revision 1.5. Technical Specification, UNIX International, Inc., Parsipanny, New Jersey, December 10 1992. http://www.openss7.org/docs/tpi.pdf.

[TPI99] Transport Provider Interface (TPI) Specification, Revision 2.0.0, Draft 2. Technical Specification, The Open Group, Parsipanny, New Jersey, 1999. http://-www.opengroup.org/onlinepubs/.

[VS90] Ian Vessey and Glen Skinner. Implementing Berkeley Sockets in System V Release 4. In *Proceedings of the Winter 1990 USENIX Conference*. USENIX, 1990.

[XNS99] Network Services (XNS), Issue 5.2, Draft 2.0. Open Group Publication, The Open Group, 1999. http://-www.opengroup.org/onlinepubs/.

[XTI99] XOpen Tranport Interface (XTI). Technical Standard XTI/TLI Revision 1.0, X Programmer's Group, 1999. http://www.opengroup.org/onlinepubs/.

A Netperf Benchmark Script

Following is a listing of the `netperf_benchmark` script used to generate raw data points for analysis:

```
#!/bin/bash
set -x
(
  sudo killall netserver
  sudo netserver >/dev/null </dev/null 2>/dev/null &
  sleep 3
  netperf_udp_range -x /dev/udp2 \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  netperf_udp_range \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  netperf_udp_range -x /dev/udp \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  sudo killall netserver
) 2>&1 | tee `hostname`.`date -uIminutes`.log
```

B Raw Data

Following are the raw data points captured using the `netperf_benchmark` script:

Table 1 lists the raw data from the `netperf` program that was used in preparing graphs for Fedora Core 6 (i386) on Porky.

Table 2 lists the raw data from the `netperf` program that was used in preparing graphs for CentOS 4 on Porky.

Table 3 lists the raw data from the `netperf` program that was used in preparing graphs for SuSE OSS 10 on Porky.

Table 4 lists the raw data from the `netperf` program that was used in preparing graphs for Ubuntu 6.10 on Porky.

Table 5 lists the raw data from the `netperf` program that was used in preparing graphs for RedHat 7.2 on Pumbah.

Table 6 lists the raw data from the `netperf` program that was used in preparing graphs for Fedora Core 6 (x86_64) HT on Daisy.

Table 7 lists the raw data from the `netperf` program that was used in preparing graphs for SuSE 10.0 OSS on Mspiggy.

Message	XTIoS		XTI		Sockets	
Size	Tx	Rx	Tx	Rx	Tx	Rx
1	714927	714928	947084	947085	740775	728170
2	717371	717372	934792	934793	745202	732710
4	713453	713454	938505	938506	750541	730419
8	704000	704001	935024	935025	745011	724798
16	697051	697052	930898	930899	746454	731250
32	688597	688598	931763	931764	748286	731657
64	686784	686785	939694	939695	740980	722478
128	674447	674448	930575	930576	742196	723733
256	657051	657052	907451	907452	740007	717115
512	651677	651678	902984	902985	718341	708200
1024	619363	619364	868516	868517	712384	693917
2048	559866	559867	793259	793260	684433	674277
4096	459220	459221	706605	706606	629194	612532
8192	367311	367312	627682	627683	554245	541436
16384	249573	249574	469472	469473	446906	437599

Table 1: FC6 on Porky Raw Data

Message	XTIoS		XTI		Sockets	
Size	Tx	Rx	Tx	Rx	Tx	Rx
1	849555	849556	1167336	1167337	861219	860982
2	845106	845107	1171086	1171087	860981	860257
4	848669	848670	1171198	1171199	863027	862307
8	828520	828521	1158247	1158248	859350	858899
16	835946	835947	1163405	1163406	856881	856418
32	837624	837625	1145328	1145329	861550	861133
64	824114	824115	1156624	1156625	850320	849599
128	811344	811345	1160676	1160677	847531	846980
256	813958	813959	1154616	1154617	842601	841396
512	804584	804585	1164623	1164624	833461	832452
1024	767812	767813	1118676	1118677	808018	806991
2048	693760	693761	1050507	1050508	766594	765236
4096	561885	561886	920261	920262	682312	681197
8192	437609	437610	678034	678035	598846	597855
16384	268808	268809	590358	590359	478197	477303

Table 2: CentOS 4 on Porky Raw Data

Message	XTIoS		XTI		Sockets	
Size	Tx	Rx	Tx	Rx	Tx	Rx
1	573781	573782	713504	713505	594660	594467
2	567733	567734	720039	720040	587883	587791
4	569997	569998	729645	729646	589438	589229
8	567197	567198	734516	734517	589559	589416
16	568657	568658	686428	686429	593745	593600
32	571096	571097	689929	689930	594827	594671
64	570663	570664	705258	705259	593679	593128
128	567062	567063	706918	706919	592829	592829
256	568372	568373	716627	716628	585737	585338
512	565382	565383	675129	675130	581023	580381
1024	546251	546252	633631	633632	576955	576220
2048	510822	510823	627276	627277	556534	555734
4096	437420	437421	577926	577927	518700	517611
8192	353468	353469	528576	528577	458838	458081
16384	258953	258954	455257	455258	378575	377998

Table 3: SuSE OSS 10 on Porky Raw Data

Message	XTIoS		XTI		Sockets	
Size	Tx	Rx	Tx	Rx	Tx	Rx
1	529545	529546	662574	662575	615243	615243
2	529833	529834	662749	662750	615219	615219
4	529409	529410	662601	662602	614769	614769
8	526374	526375	652110	652111	614941	614941
16	527462	527463	654046	654047	614494	614494
32	525083	525084	649961	649962	614532	614532
64	524388	524389	648902	648903	613586	613586
128	521954	521955	650092	650093	612867	612867
256	508588	508589	644845	644846	598102	598102
512	505348	505349	642097	642098	595758	595758
1024	481918	481919	623680	623681	590474	590474
2048	451341	451342	600956	600957	568011	568011
4096	390587	390588	552289	552290	529874	529874
8192	304485	304486	499277	499278	466069	466069
16384	232667	232668	405488	405489	391741	391741

Table 4: Ubuntu 6.10 on Porky Raw Data

Message Size	XTIoS Tx	XTIoS Rx	XTI Tx	XTI Rx	Sockets Tx	Sockets Rx
1	1133043	1133044	1560516	1560517	1422429	1422429
2	1136533	1136534	1562461	1562462	1418493	1418493
4	1136695	1136696	1578993	1578994	1415739	1415129
8	1142312	1142313	1578110	1578111	1415738	1415129
16	1139623	1139624	1571645	1571646	1412013	1411527
32	1140659	1140660	1573956	1573957	1418429	1418429
64	1136007	1136008	1574064	1574065	1406332	1406332
128	1106231	1106232	1541064	1541065	1370828	1370828
256	1073676	1073677	1535408	1535409	1358240	1357444
512	1026932	1026933	1517692	1517693	1299434	1299434
1024	941481	941482	1455261	1455262	1211158	1211158
2048	793802	793803	1351690	1351691	1073543	1073543
4096	610252	610253	1216734	1216735	872281	872281
8192	416164	416165	1033488	1033489	644953	644953
16384	248762	248763	780198	779901	419478	419478

Table 5: RedHat 7.2 on Pumbah Raw Data

Message Size	XTIoS Tx	XTIoS Rx	XTI Tx	XTI Rx	Sockets Tx	Sockets Rx
1	553383	553384	1009820	1009820	731713	731713
2	550020	550021	1005658	1005659	726596	726596
4	549600	549601	993347	993348	733634	733634
8	549073	549074	1000195	1000196	724320	724320
16	549514	549515	1000525	1000526	725440	725440
32	548447	548447	1007185	1007186	728707	728707
64	545329	545330	994739	994740	720612	720612
128	540519	540520	999002	999003	722801	722801
256	521171	521172	994474	994475	723606	723606
512	508589	508590	982028	982029	709207	709207
1024	483899	483900	951564	951565	707136	707136
2048	446004	446005	897395	897396	688775	688775
4096	387509	387510	795327	795328	650128	650128
8192	302141	302142	677573	677573	605011	605011
16384	211149	211150	505129	505130	503729	503729

Table 6: Fedora Core 6 (x86_64) HT on Daisy Raw Data

Message Size	XTIoS Tx	XTIoS Rx	XTI Tx	XTI Rx	Sockets Tx	Sockets Rx
1	479564	479565	591461	591462	482975	481652
2	480678	480679	592805	592806	481606	480276
4	478366	478367	593255	593256	480746	479680
8	473615	473616	589930	589931	479021	477301
16	471973	471974	585814	585815	478449	476241
32	474980	474981	585272	585273	480508	478812
64	466618	466619	587244	587245	474745	472577
128	465623	465624	582449	582450	472031	470381
256	458158	458159	587534	587534	466018	463747
512	446356	446357	586409	586410	450769	448312
1024	421072	421073	567213	567214	435038	433157
2048	368990	368991	543818	543819	397745	395329
4096	290402	290403	500380	500381	344058	341942
8192	218918	218919	438956	438957	265907	264098
16384	137005	137006	348956	348957	192224	191737

Table 7: SuSE 10.0 OSS on Mspiggy Raw Data