STREAMS vs. Sockets Performance Comparison for SCTP

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* TCP implementation and STREAMS TPI SCTP and XTIoS TCP 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 System 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] directly or without reconfiguration.

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 TCP (`tcp`*).*
: 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 TCP. While the implementation uses the Transport Provider Interface and STREAMS to communicate with the driver, internal to the driver a TCP Socket is opened and conversion between STREAMS and Sockets performed.

OpenSS7 STREAMS TPI SCTP Driver (`sctp`*).*
: A STREAMS pseudo-device driver that fully implements SCTP and communicates with the IP layer in the kernel. Both the OpenSS7 native sockets version and the STREAMS version are based on the same protocol engine core.

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

3.1 Linux Kernel TCP

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.

Both Linux TCP 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

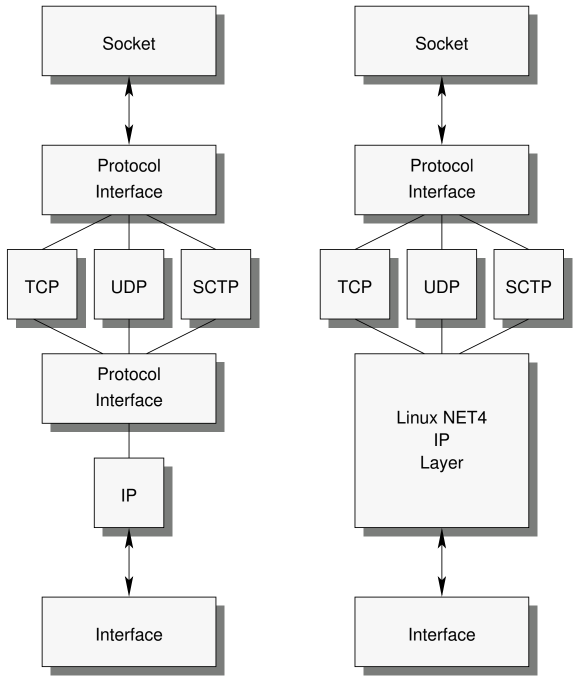

Figure 1: Sockets: BSD and Linux

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]

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 TCP Sockets 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 TCP Sockets 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, under-performing 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,

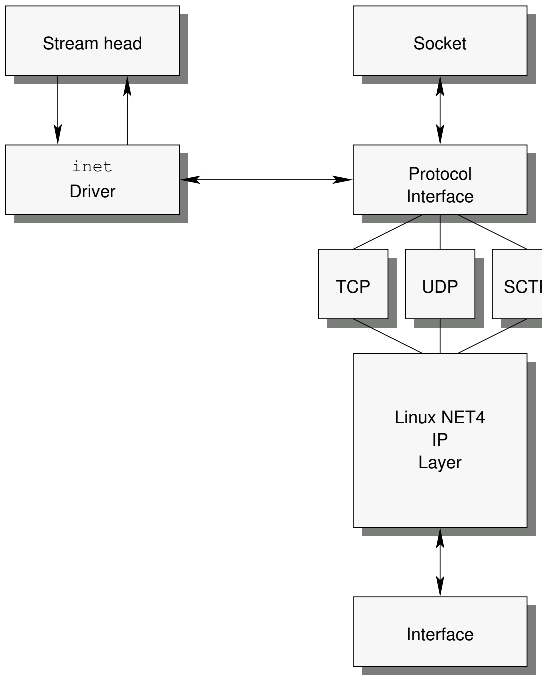

Figure 2: XTI over Sockets `inet` Driver

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]

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 SCTP is a direct STREAMS implementation that uses the `sctp` 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

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.

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.

presents the Transport Provider Interface (TPI) [TPI99] for use by upper level modules and the XTI library [XTI99].

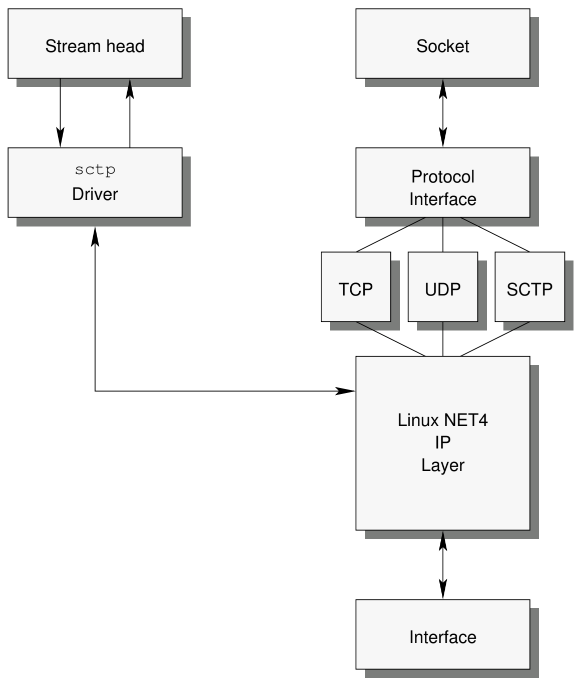

Figure 3: STREAMS `sctp` Driver

Linux Fast-STREAMS also provides a raw IP driver (`raw`) and an SCTP driver (`sctp`) that operate in the same fashion as the `sctp` 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, two benchmark scripts (called `netperf_benchmark` and `netperf_nice2`) were 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 these scripts are provided in *Appendix A*.

4.1 Implementations Tested

The following implementations were tested:

TCP Sockets. This is the Linux NET4 Sockets implementation of TCP, described in *Section 3.1*, with normal scheduling priorities. Normal scheduling priority means invoking the sending and receiving processes without altering their run-time scheduling priority.

TCP Sockets with artificial process priorities. This is the Linux NET4 Sockets implementation of TCP, described in *Section 3.1*, with artificial scheduling priorities. Artificial scheduling priority means invoking the sending and receiving processes with extreme run-timer scheduling priority alterations.

Early on in the testing it was discovered that Sockets exhibits remarkably poor performance on UP machines when the process priority of the sender and the receiver are the same. This is largely due to deficiencies in the traditional Socket implementation with regard to buffering, flow control and scheduling. When the receiver has an equal or higher effective priority than the sender, Sockets causes context thrashing[9] between the sender and receiver that impairs overall performance.

To circumvent these deficiencies, a test was also performed by increasing the priority of the sender to the maximum (nice -n -20) and decreasing the priority of the receiver to the minimum (nice -n 19). While this is a markedly artificial situation, it is indicative of the performance that could be achieved by Sockets if it did not have these fundamental deficiencies.

These results are not indicative of the performance that can be expected by most TCP applications on the loop-back interface.

STREAMS XTIoS TCP. This is the OpenSS7 STREAMS implementation of XTI over Sockets for TCP, 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 SCTP 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 TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

Delay. *Figure 5* plots the average message delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

Throughput. *Figure 6* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

9. In fact, it appears that under normal conditions on the loopback interface, one context switch per *write* is occuring.

Improvement. *Figure 7* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes. Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

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 TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

Delay. *Figure 9* plots the average message delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

Throughput. *Figure 10* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

Improvement. *Figure 11* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes.

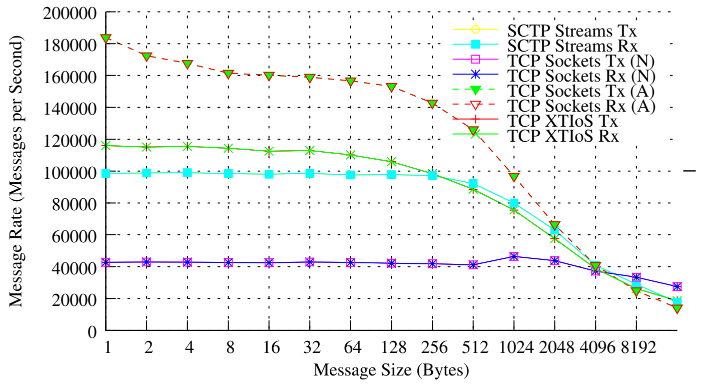

Figure 4: Fedora Core 6 on Porky Performance

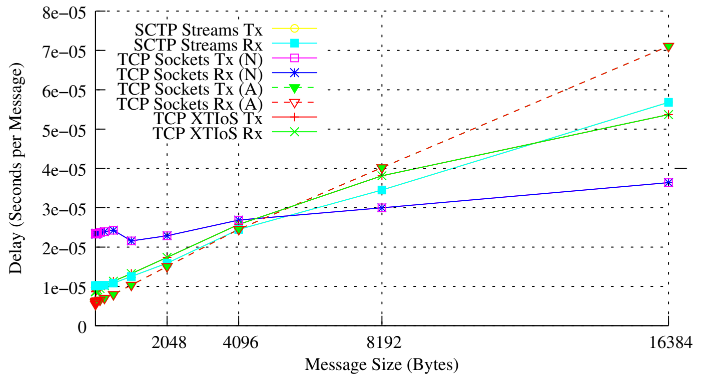

Figure 5: Fedora Core 6 on Porky Delay

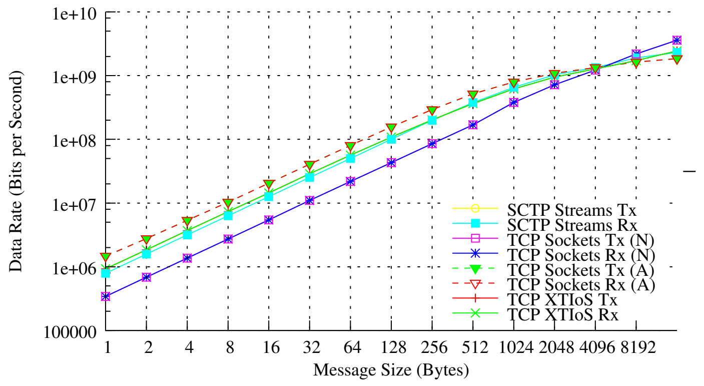

Figure 6: Fedora Core 6 on Porky Throughput

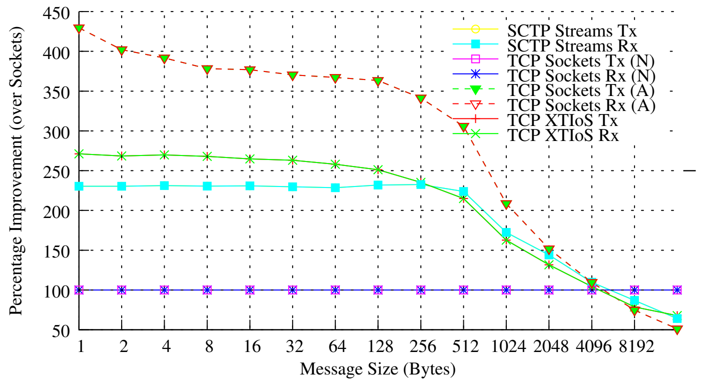

Figure 7: Fedora Core 6 on Porky Comparison

Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

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 TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

Delay. *Figure 13* plots the average message delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

Throughput. *Figure 14* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

Improvement. *Figure 15* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes. Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

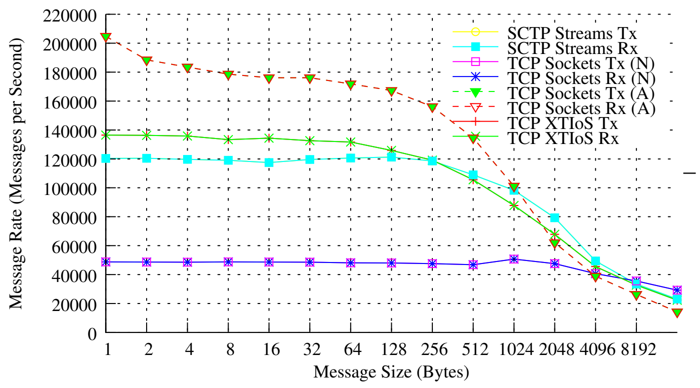

Figure 8: CentOS on Porky Performance

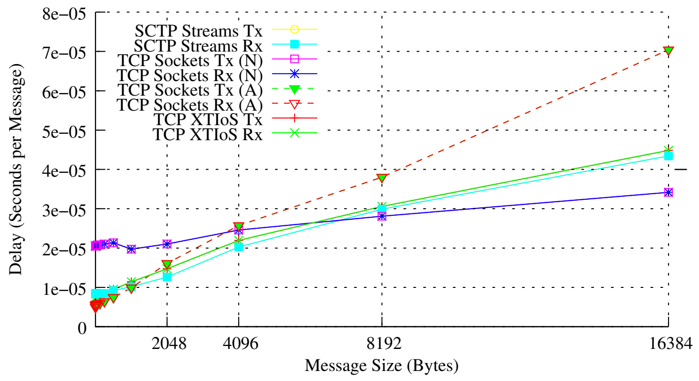

Figure 9: CentOS on Porky Delay

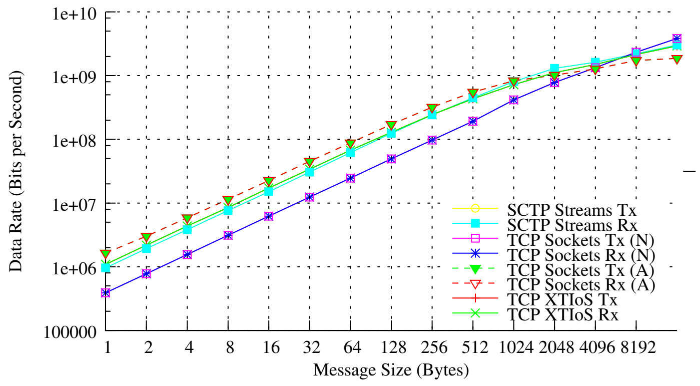

Figure 10: CentOS on Porky Throughput

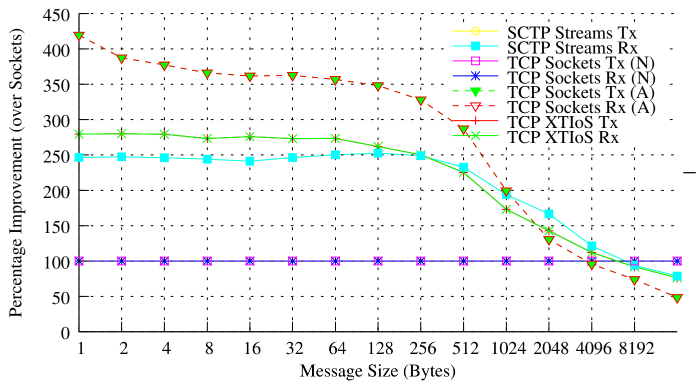

Figure 11: CentOS on Porky Comparison

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.

5.1.4 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 16* plots the measured performance of TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

Delay. *Figure 17* plots the average message delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

Throughput. *Figure 18* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

Improvement. *Figure 19* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes. Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

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

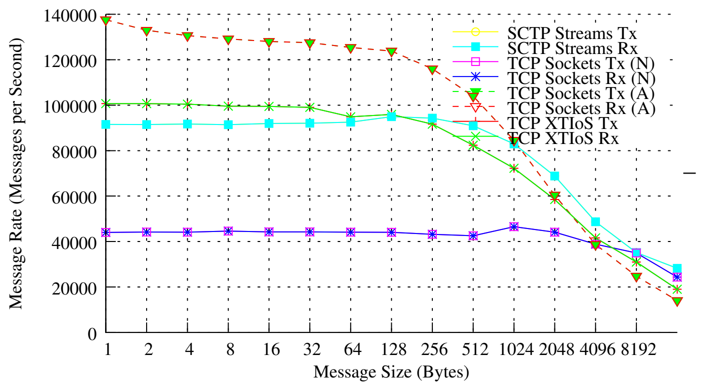

Figure 12: SuSE on Porky Performance

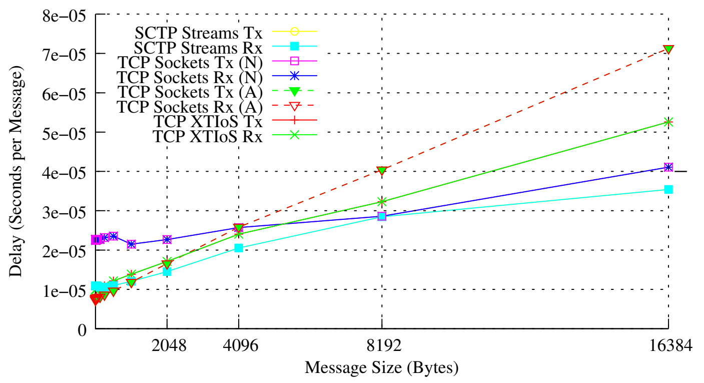

Figure 13: SuSE on Porky Delay

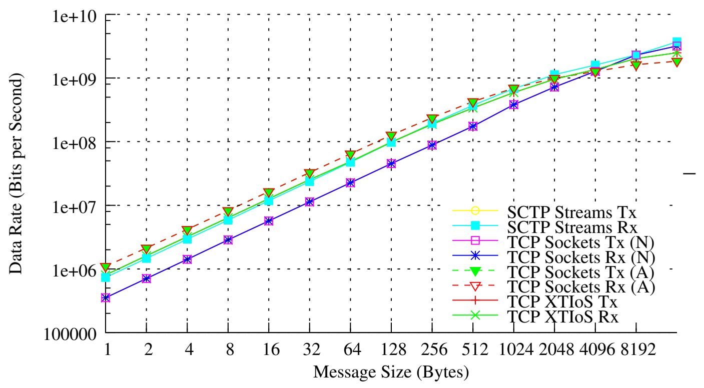

Figure 14: SuSE on Porky Throughput

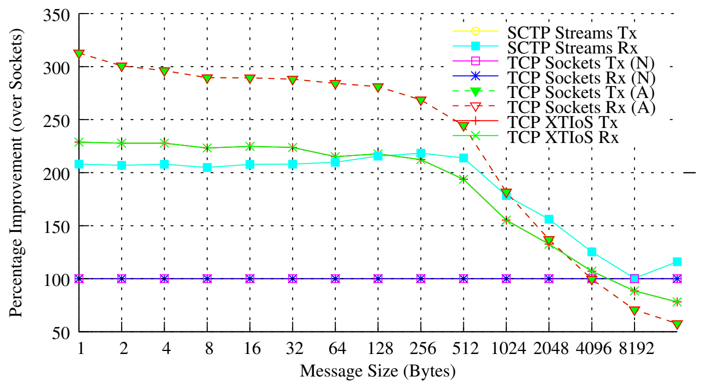

Figure 15: SuSE on Porky Comparison

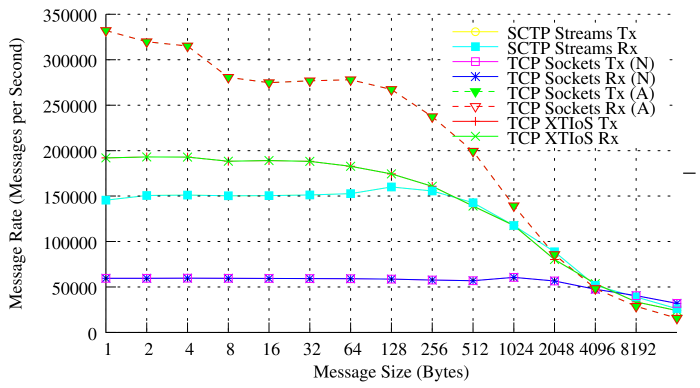

Figure 16: Ubuntu 7.04 on Porky Performance

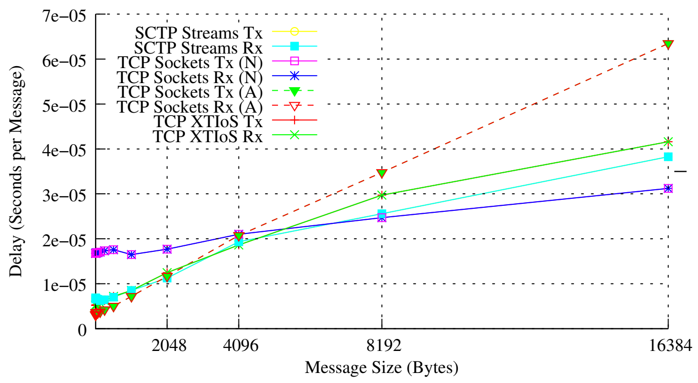

Figure 17: Ubuntu 7.04 on Porky Delay

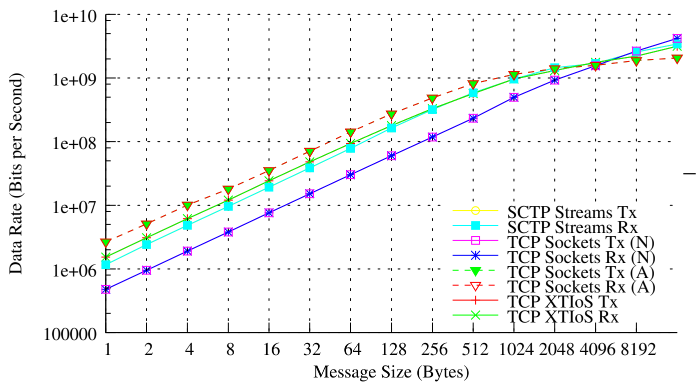

Figure 18: Ubuntu 7.04 on Porky Throughput

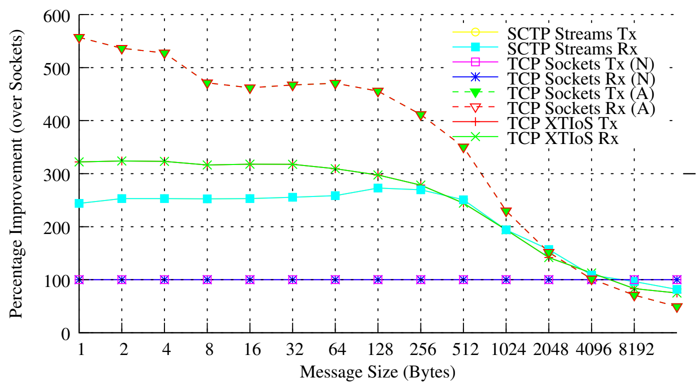

Figure 19: Ubuntu 7.04 on Porky Comparison

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 20* plots the measured performance of TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

STREAMS demonstrates significant improvements at mssage sizes of less than 1024 bytes, and comparable performace at larger message sizes.

A significant result is that the TCP XTI over Socket approach indeed provided improvements over TCP Sockets itself at message sizes beneath 1024 bytes. This improvement can only be accounted for by buffering and schedule differences, and when the receiving process was given a lower scheduling priority than the sending process, TCP Sockets performed much better.

Delay. *Figure 21* plots the average message delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

STREAMS demonstrates significant improvements at message sizes of less than 1024 bytes, and comparable performance at larger message sizes.

The slope of the dleay curve is best for SCTP XTI, then TCP Sockets (for message sizes greater than or equal to 1024 bytes), then TCP XTI over Sockets, then TCP Sockets (with low priority receiver).

The slope of the delay curve either indicates that SCTP XTI STREAMS has the best overall per-byte handling performance.

Throughput. *Figure 22* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

STREAMS demonstrates significant improvements at most message sizes.

As can be seen from *Figure 22*, all implementations exhibit strong power function characteristics, indicating structure and robustness for each implementation, except for TCP Sockets at regular scheduling prioritys.

TCP Sockets at regular scheduling priorities exhibits a strong dicontinuity between message sizes of 512 btyes and 1024 bytes. This non-linearity can be explained by the poor buffering, flow control and scheduling capabilities of Sockets when compared to STREAMS. Indeed, when the receiving process was artificially downgraded to a low priority (nice -19) to avoid the weaknesses inherent in the Sockets approach, it exhibits better characteristics. Perhaps surprisingly, by wrapping the internal socket with STREAMS, the TCP XTIoS approach does not exhibit the weaknesses of Sockets alone and in some way compensates for the defficiencies of Socket buffering, flow control and scheduling.

Improvement. *Figure 23* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes. Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

STREAMS demonstrates significant improvements (approx. 150-180% improvement) at message sizes below 1024 bytes. That STREAMS SCTP and TCP give such an improvement over a wide range of message sizes is a dramatic improvement. Note that dramatic improvement is also demonstrated for TCP Sockets when the receiver is artificially given the lowest possible scheduling priority, thus circumventing Sockets' poor scheduling and flow control characteristics.

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

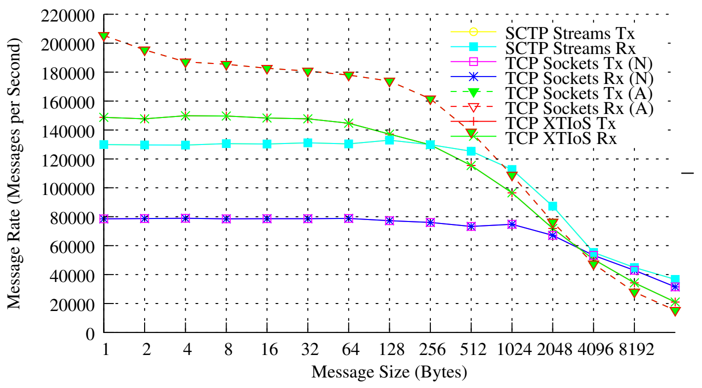

Figure 20: RedHat 7.2 on Pumbah Performance

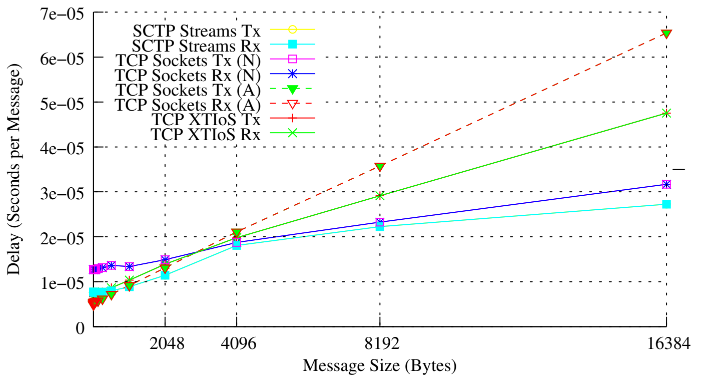

Figure 21: RedHat 7.2 on Pumbah Delay

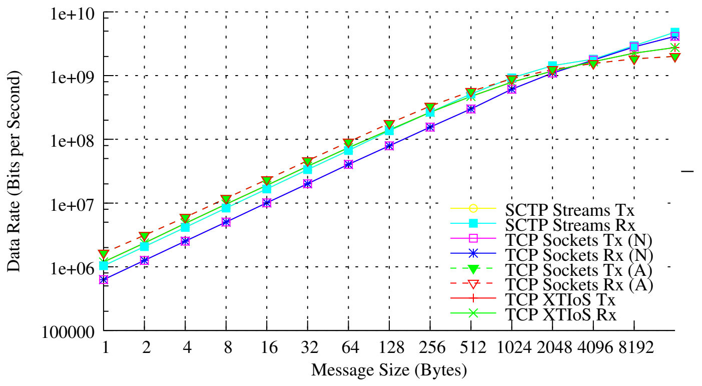

Figure 22: RedHat 7.2 on Pumbah Throughput

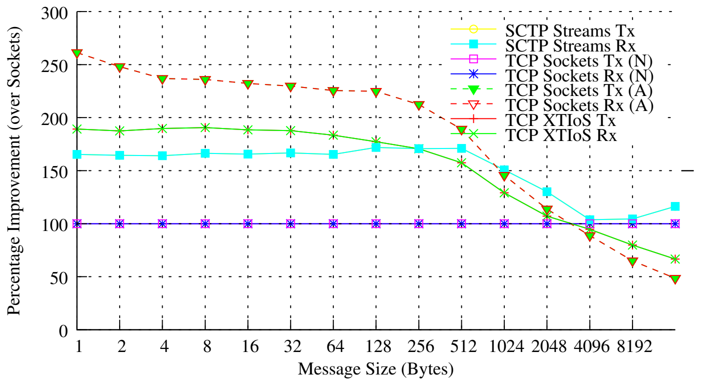

Figure 23: RedHat 7.2 on Pumbah Comparison

patches. This is the `x86_64` distribution with recent updates.

Performance. *Figure 24* plots the measured performance of TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

Delay. *Figure 25* plots the average message delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

Throughput. *Figure 26* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

Improvement. *Figure 27* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes. Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

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.

Performance. *Figure 28* plots the measured performance of TCP Sockets (both normal and artificial scheduling priorities), TCP XTIoS STREAMS and SCTP XTI STREAMS

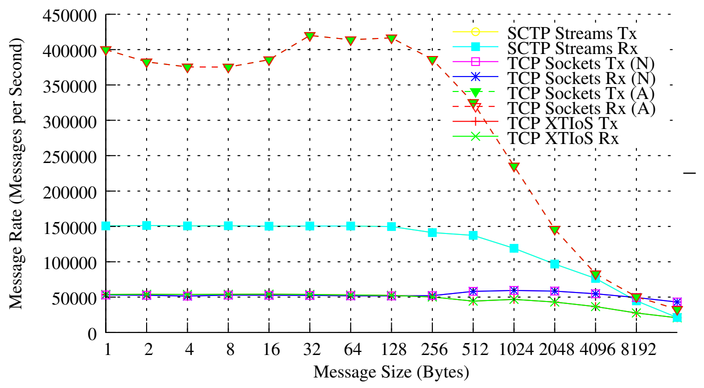

Figure 24: Fedora Core 6 on Daisy Performance

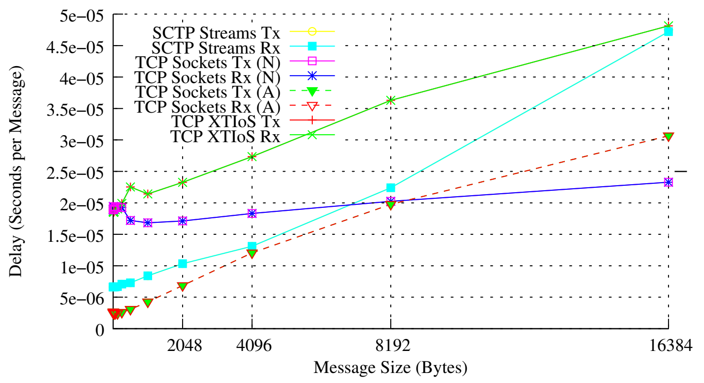

Figure 25: Fedora Core 6 on Daisy Delay

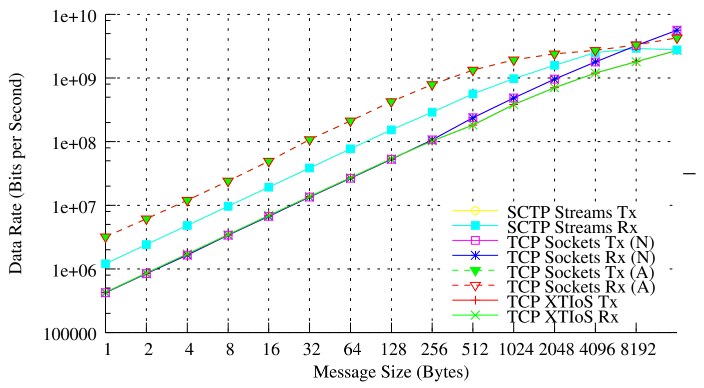

Figure 26: Fedora Core 6 on Daisy Throughput

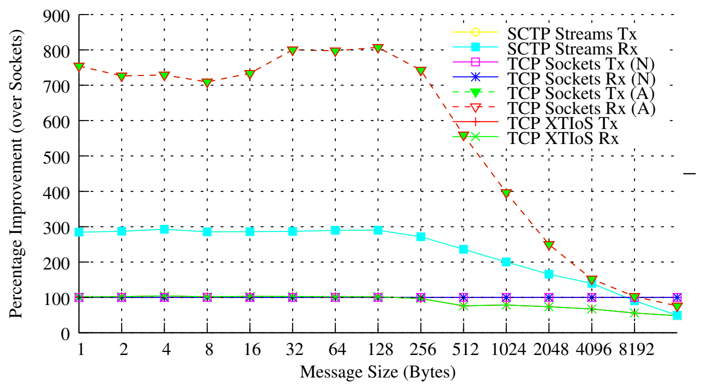

Figure 27: Fedora Core 6 on Daisy Comparison

implementations. The higher performing TCP Sockets graph (with dashed lines and designated with '(A)') is the artifical scheduling priority plot. The under performing TCP Sockets graph (with the solid lines and designated with '(N)') is the normal scheduling plot.

TCP Sockets with normal scheduling shows dismal performance in comparison to both STREAMS – TCP XTIoS and SCTP XTI – approaches at all message sizes beneath 4096 bytes. It is necessary to artificially reduce the receiver priority to a minimum (`nice -n 19`) and artificially increase the sender priority to a maximum (`nice -n -20`) to acheive better results on TCP Sockets beneath 4096 bytes, at the cost of poorer performance above 4096 bytes.

The slightly different performance between TCP XTIoS and SCTP XTI can be explained by the significant overheads that the SCTP protocol introduces on small message sizes.

STREAMS demonstrates significant improvements at message sizes of less than 4096 bytes, and improvements at all message sizes.

A significant result is that the TCP XTI over Socket approach indeed provided improvements over TCP Sockets itself at message sizes beneath 4096 bytes. This improvement can only be accounted for by buffering differences. When the receiving process was given a lower scheduling priority (`nice -n 19`) than the sending process (`nice -n -20`), forcing the implementation into a tight corner, TCP Sockets performed better; however, only for smaller message sizes.

Delay. *Figure 29* plots the average delay of TCP Sockets (both normal and artificial), TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

The average delay of TCP XTIoS STREAMS and SCTP XTI STREAMS approaches is similar and comparable with TCP Sockets with artificial scheduling and message sizes beneath 4096. With normal scheduling, however, TCP Sockets has poor per message delays (intercept) but superior per-byte delays (slope).

Throughput. *Figure 30* plots the effective throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

All curves exhibit good power function characteristics beneath 1024 byte message sizes, indicating structure and robustness for each implementation, but each implementation exhibits saturation characteristics above 1024 bytes.

Improvement. *Figure 31* plots the relative percentage of throughput of TCP Sockets, TCP XTIoS STREAMS and SCTP XTI STREAMS implementations.

For the normal case, TCP XTIoS STREAMS and SCTP XTI STREAMS exhibit significant improvements over TCP Sockets for message sizes less than 4096 bytes and are superior or comparable at message sizes greater than 4096 bytes. Forcing TCP Sockets into a specific behaviour by artificially maximizing the sender priority and minimizing the receive priority results in improved behaviour below 4096 bytes for TCP Sockets but is worse than normal scheduling priority TCP Sockets for message sizes of 4096 bytes or more.

STREAMS demonstrates significant improvements (approx. 250% improvement) at message sizes below 1024 bytes. That STREAMS SCTP gives a 200% improvement over a wide range of message sizes is a dramatic improvement.

6 Analysis

With some caveats as described at the end of this section, the results are consistent enough across the various distributions and

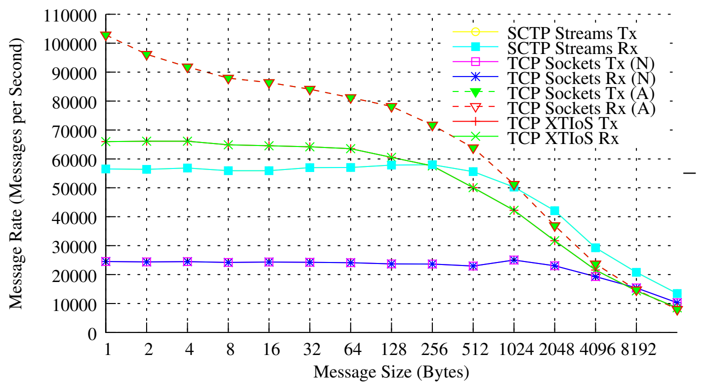

Figure 28: SuSE 10.0 OSS Mspiggy Performance

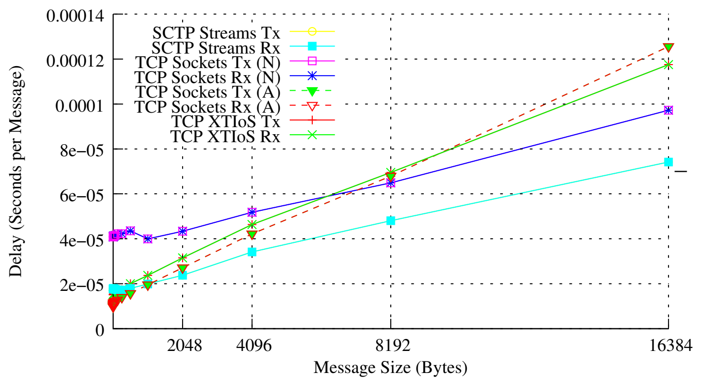

Figure 29: SuSE 10.0 OSS Mspiggy Delay

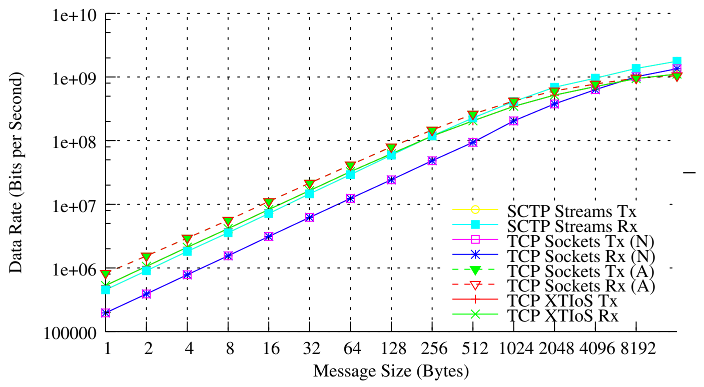

Figure 30: SuSE 10.0 OSS Mspiggy Throughput

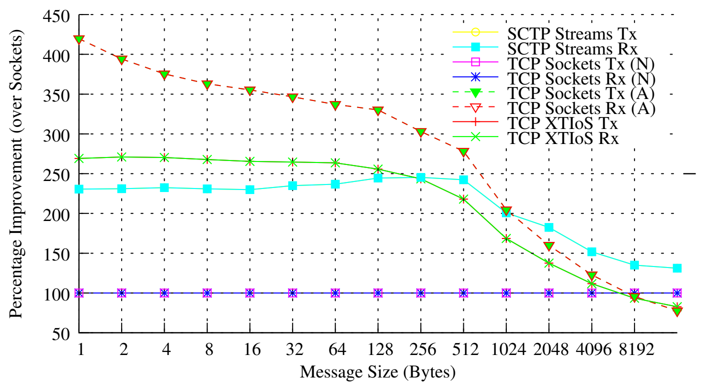

Figure 31: SuSE 10.0 OSS Mspiggy Comparison

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 SCTP is superior to that of the Linux Kernel Sockets implementation of TCP. In fact, STREAMS TPI SCTP implementation performance at smaller message sizes is significantly (as much as 200-300%) greater than that of Linux Kernel Sockets TCP. As the common belief is that STREAMS would exhibit poorer performance, this is perhaps a startling result to some.

Perhaps even more surprising is that the STREAMS implementation of TCP using XTI over Sockets is superior to the TCP Socket alone! And, again by as much as 200-300%.

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

Send processing. When Linux Sockets TCP 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 TCP send operation. Linux 2.4 kernels are 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 Sockets on 2.6 and 2.4 kernels.[10]

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 SCTP driver utilizes this feature (but the STREAMS XTIoS TCP driver cannot). STREAMS also has the feature that a write offset can be applied to all data block allocated and passed downstream. However, neither the STREAMS TPI SCTP nor XTIoS TCP drivers use this capability. It is currently only used by the second generation STREAMS UDP and RAWIP drivers.

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 SCTP 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 TCP, 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 SCTP, 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 SCTP 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 TCP, the same `skbuff` is reused on both sides of the interface. For STREAMS SCTP, 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 TCP, 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 SCTP implementation on the receive side, again there is no `sock` structure, so the socket locking and backlog techniques performed by SCTP at the lower layer do not apply. When the STREAMS SCTP 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 TCP, 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

10. For example, Fedora Core 6 on Porky churns out 12,000 packets per second at 1 byte (*Figure 4*), whereas RedHat 7.2 on Pumbah churns out 15,000 packets per second at 1 byte (*Figure 20*).

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 charactersitics of the loop-back interface make it somewhat different from regular network interfaces:

1. Loop-back interfaces do not require checksums.

   One of the major disadvantages of SCTP over TCP from a protocol performance perspective is the increased cost of the CRC-32C checksum used by SCTP over the Fletcher32 checksum used by TCP. Using the loop-back interface avoids this checksum comparison as both TCP and SCTP do not perform checksum on loop-back.

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.

   This, in fact, provides an advantage to TCP. Even a light degree of packet loss impacts TCP's performance to a far greater degree than SCTP.

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.

6.2.2 Effective Bandwidth

Tests compare performance of two rather different implementations of TCP against a single implementation of SCTP. TCP and SCTP have inherent differences in the protocol that affect the efficiency at various load points.

For example, whereas TCP can coallesce many small writes into a single contiguous segment for transmission in a single TCP packet, SCTP must normally create individual data chunks for each write. Some of the original `iperf` testing on the OpenSS7 Linux Native Sockets version of SCTP used a specialized `SOCK_STREAM` mode that ignored message boundaries and only supported one SCTP stream. This is a far better comparison to TCP in this respect. The `netperf` package also takes this approach by setting the `T_MORE` bit on all calls to `t_snd(3)`.

Also, when message sizes are small, SCTP normally has significant overheads in the protocol that consume available bandwidth and reduce efficiency. For example, for messages of N bytes, transmitted on the loop-back interface, for TCP this consists of the IP header, the TCP header, and the data; for SCTP, the IP header, the SCTP header, and one data chunk header (plus the padding to pad the data to the next 32-bit boundary) for each message bundled. Again, `netperf` sets the `T_MORE` bit on all calls to `t_snd(3)` in an attempt to behave more like TCP for comparison testing.

7 Conclusions

These experiments have shown that the *Linux Fast-STREAMS* implementation of STREAMS SCTP, as well as STREAMS TCP using XTIoS, networking outperforms the Linux Sockets TCP implementation by a significant amount (approx. 200-300%).

> *The Linux Fast-STREAMS implementation of STREAMS SCTP and TCP 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 purpse), 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 TCP 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 SCTP and TCP indeed performs better than Linux Sockets TCP.

- *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 UDP 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

One script was used to generate normal data for all implementations. 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_sctp_range --mult=2 -x /dev/sctp_t \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  netperf_tcp_range  --mult=2 \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  netperf_tcp_range  --mult=2 -x /dev/tcp \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  sudo killall netserver
) 2>&1 | tee ‘hostname‘.‘date -uIminutes‘.log
```

Another script was used to generate artificial process priorities for TCP Socket data. Following is a listing of the `netperf_nice2` script used to generate raw data points for analysis:

```
#!/bin/bash
set -x
(
  sudo killall netserver
  sudo nice -n 19 netserver >/dev/null </dev/null 2>/dev/null &
  sleep 3
  sudo nice -n -20 netperf_tcp_range  --mult=2 \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  sudo nice -n -20 netperf_tcp_range  --mult=2 -x /dev/tcp \
    --testtime=10 --bufsizes=131071 --end=16384 ${1+"$@"}
  sudo nice -n -20 netperf_sctp_range --mult=2 -x /dev/sctp_t \
    --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 7.04 on Porky.

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

Table **??** 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.

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	1159849	1159849	985536	985536	427906	427906	1838370	1838370
2	1151023	1151023	988000	988000	428911	428911	1724348	1724348
4	1154867	1154867	989624	989624	428147	428147	1676809	1676809
8	1143217	1143217	983939	983939	426783	426783	1614072	1614072
16	1125275	1125275	980890	980890	424960	424960	1601554	1601554
32	1128981	1128981	985902	985902	429033	429033	1589172	1589172
64	1101200	1101200	975575	975575	426793	426793	1566994	1566994
128	1058599	1058599	977760	977760	421511	421511	1531398	1531398
256	983766	983766	972058	972058	418119	418119	1427194	1427194
512	885757	885757	922080	922080	411923	411923	1259498	1259498
1024	755090	755090	799680	799680	463988	463988	968138	968138
2048	575955	575955	629764	629764	437400	437400	663761	663761
4096	388472	388472	408505	408505	372535	372535	408223	408223
8192	262348	262348	289960	289960	333611	333611	248790	248790
16384	186284	186284	176092	176092	274833	274833	140657	140657

Table 1: FC6 on Porky Raw Data

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	1364620	1364620	1203447	1203447	488139	488139	2047905	2047905
2	1362356	1362356	1203841	1203841	486477	486477	1884331	1884331
4	1358177	1358177	1196858	1196858	486114	486114	1834986	1834986
8	1333106	1333106	1190781	1190781	487920	487920	1785688	1785688
16	1343312	1343312	1174897	1174897	486965	486965	1761354	1761354
32	1325970	1325970	1196167	1196167	485464	485464	1761113	1761113
64	1317137	1317137	1205860	1205860	481714	481714	1718639	1718639
128	1258475	1258475	1212792	1212792	480424	480424	1672210	1672210
256	1191616	1191616	1185418	1185418	476034	476034	1561106	1561106
512	1055471	1055471	1090139	1090139	468590	468590	1345495	1345495
1024	878106	878106	983178	983178	507169	507169	1011463	1011463
2048	678733	678733	793133	793133	475515	475515	620730	620730
4096	455788	455788	493888	493888	406640	406640	389039	389039
8192	326908	326908	334676	334676	355857	355857	262913	262913
16384	223135	223135	230284	230284	292729	292729	142018	142018

Table 2: CentOS on Porky Raw Data

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	1006583	1006583	915368	915368	439922	439922	1376226	1376226
2	1006782	1006782	914641	914641	442092	442092	1329413	1329413
4	1004712	1004712	917314	917314	441085	441085	1306239	1306239
8	995951	995951	914113	914113	446082	446082	1291276	1291276
16	994443	994443	919517	919517	442389	442389	1280033	1280033
32	990631	990631	920822	920822	442626	442626	1275191	1275191
64	948700	948700	925600	925600	441098	441098	1253906	1253906
128	959196	959196	949517	949517	440469	440469	1238290	1238290
256	916726	916726	942804	942804	431868	431868	1160232	1160232
512	823708	823708	909330	909330	425150	425150	1039639	1039639
1024	722228	722228	829716	829716	465298	465298	845664	845664
2048	584745	584745	688489	688489	441046	441046	604536	604536
4096	416112	416112	487168	487168	388701	388701	387125	387125
8192	309676	309676	351264	351264	349797	349797	247468	247468
16384	190270	190270	282432	282432	243425	243425	140259	140259

Table 3: SuSE on Porky Raw Data

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	1919955	1919955	1454549	1454549	596044	596044	3320562	3320562
2	1929907	1929907	1507194	1507194	596055	596055	3197381	3197381
4	1928627	1928627	1510299	1510299	597056	597056	3150297	3150297
8	1883607	1883607	1503457	1503457	595376	595376	2805778	2805778
16	1891068	1891068	1505280	1505280	594741	594741	2747410	2747410
32	1881287	1881287	1512486	1512486	592212	592212	2767590	2767590
64	1827200	1827200	1526600	1526600	590804	590804	2779697	2779697
128	1744017	1744017	1600200	1600200	586460	586460	2672735	2672735
256	1607224	1607224	1555750	1555750	577130	577130	2374032	2374032
512	1390792	1390792	1427619	1427619	569271	569271	1993970	1993970
1024	1176572	1176572	1176774	1176774	606248	606248	1393740	1393740
2048	801656	801656	887619	887619	565686	565686	857393	857393
4096	536607	536607	517264	517264	475933	475933	482003	482003
8192	336493	336493	390888	390888	404821	404821	287576	287576
16384	240334	240334	261424	261424	320299	320299	157594	157594

Table 4: Ubuntu on Porky Raw Data

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	1487391	1487391	1299066	1299066	785896	785896	2053453	2053453
2	1477331	1477331	1296171	1296171	787870	787870	1954151	1954151
4	1498000	1498000	1295513	1295513	789527	789527	1870593	1870593
8	1496916	1496916	1306058	1306058	785417	785417	1853853	1853853
16	1482747	1482747	1302501	1302501	786693	786693	1826834	1826834
32	1476759	1476759	1311354	1311354	786711	786711	1806797	1806797
64	1446800	1446800	1303624	1303624	788717	788717	1779218	1779218
128	1370687	1370687	1329321	1329321	773130	773130	1738258	1738258
256	1297649	1297649	1298275	1298275	760501	760501	1615250	1615250
512	1153941	1153941	1252839	1252839	732871	732871	1385621	1385621
1024	965640	965640	1126673	1126673	747633	747633	1088448	1088448
2048	718914	718914	873625	873625	671029	671029	763770	763770
4096	504928	504928	552977	552977	533349	533349	473762	473762
8192	343355	343355	449116	449116	430065	430065	279581	279581
16384	210356	210356	366916	366916	315475	315475	152971	152971

Table 5: RedHat 7.2 on Pumbah Raw Data

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	537894	537894	1507754	1507754	530271	530271	4000488	4000488
2	538587	538587	1513008	1513008	526868	526868	3827553	3827553
4	536886	536886	1507753	1507753	515001	515001	3754113	3754113
8	539140	539140	1509625	1509625	529006	529006	3753260	3753260
16	540883	540883	1502556	1502556	525746	525746	3857426	3857426
32	537277	537277	1504196	1504196	524605	524605	4199442	4199442
64	529188	529188	1504607	1504607	519086	519086	4139794	4139794
128	524924	524924	1496666	1496666	516056	516056	4166290	4166290
256	502111	502111	1412554	1412554	520030	520030	3861920	3861920
512	443904	443904	1373139	1373139	581477	581477	3252110	3252110
1024	467082	467082	1190510	1190510	593941	593941	2353643	2353643
2048	429993	429993	967822	967822	584436	584436	1460241	1460241
4096	365776	365776	763228	763228	546152	546152	829584	829584
8192	275454	275454	446314	446314	493871	493871	505873	505873
16384	207719	207719	211762	211762	429703	429703	326267	326267

Table 6: Fedora Core 6 on Daisy Raw Data

Msg	TCP XTIoS		SCTP XTI		Sockets (N)		Sockets (A)	
Size	Tx	Rx	Tx	Rx	Tx	Rx	Tx	Rx
1	659556	659556	565257	565257	245146	245146	1028581	1028581
2	661049	661049	563824	563824	244002	244002	961781	961781
4	661009	661009	568400	568400	244579	244579	918068	918068
8	648681	648681	559273	559273	242212	242212	879112	879112
16	645369	645369	559123	559123	243221	243221	863951	863951
32	641812	641812	569891	569891	242555	242555	840722	840722
64	635102	635102	570400	570400	240886	240886	811724	811724
128	605405	605405	578742	578742	236768	236768	781586	781586
256	575741	575741	580136	580136	236547	236547	716701	716701
512	500477	500477	556189	556189	229565	229565	638458	638458
1024	422162	422162	502356	502356	250385	250385	512218	512218
2048	317378	317378	420964	420964	230706	230706	369249	369249
4096	215656	215656	292934	292934	193080	193080	237092	237092
8192	143879	143879	208137	208137	154140	154140	147300	147300
16384	85080	85080	134818	134818	102834	102834	79644	79644

Table 7: SuSE 10.0 OSS on Mspiggy Raw Data