Implementing POSIX Sockets for Linux Fast-STREAMS

Design for Linux

Brian F. G. Bidulock[*]
OpenSS7 Corporation

June 16, 2007

Abstract

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 Layer Interface (TLI) [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 the *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 (in 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] without system reconfiguration.

UNIX System V Release 4.2 supported four Application Programming 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 SVR3 and SVR4, later standardized by *X/Open* as the *XTI* described below. This interface operated differently than the XTI in subtle ways, and 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*[TPI99].

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 the 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, operated differently from the POSIX interface in subtle ways, and is now deprecated in favour of the POSIX/SUS standard Sockets interface.

POSIX Sockets. Sockets were standardized by X/Open, later the OpenGroup,[3] and IEEE in the POSIX standardization process. They appear in XNS 5.2 [XNS99], SUSv1 [SUS95], SUSv2 [SUS98] and SUSv3 [SUS03]. POSIX/SUS Sockets is now the common application environment for accessing networking, deprecating the XTI for TCP/IP networking applications.

[*]bidulock@openss7.org

1. AIX, for example.
2. HP-UX, for example.
3. *http://www.opengroup.org/*

One systems supporting STREAMS, but not traditionally supporting Sockets (such as SVR4), there are a number of approaches toward supporting BSD and POSIX Sockets:

Compatibility Library. Under this approach, the oldest of approaches for STREAMS, 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 abandonned 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 (`sockmod`) is pushed on a Transport Provider Interface (TPI) stream. The library (`socklib`) and cooperating module (`sockmod`) provide the BBN or POSIX Socket API [VS90] [Mar01]. This is the intermediate implementation approach taken by SVR4 and earlier releases of Solaris.

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 a TPI-based STREAMS transport provider, it is just that system calls instead of library calls are used to implement the interface [Mar01]. This is another intermediate implementation approach taken by later releases of Solaris.

System Calls. Under this approach, even the socket(3) call is moved into the kernel. Conversion between POSIX/BSD Sockets 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, domains and protocols [Mar01].

1.2 Sockets

Sockets were originally developed as part of the BBN DARPA contract for providing TCP/IP networking for 4BSD UNIX. The TCP/IP networking stack was first implemented on 4.1aBSD and included in 4.2BSD by the CSRG [MBKQ97]. The Sockets interface had the objective of being a general purpose, network agnostic, interprocess communications system. The first networking components implemented using Sockets were TCP/IP (released in 4.2BSD), XNS (released in 4.3BSD) and ISO (released in 4.4BSD).

Although the networking subsystems for BSD Sockets were intended to be general purpose, only the TCP/IP networking components have received wide use.

Systems that take the BSD approach to networking seldom support STREAMS.[4] For systems traditionally supporting Sockets and then retrofitted to support the XTI interface, there is one approach toward supporting XTI without retrofitting the entire networking stack to support STREAMS:[5]

XTI Compatibility Library. Several implementations of XTI on UNIX utilize the concept of an XTI compatibility library.[6] This is purely a shared object library approach to providing XTI. Under this approach it is possible to use the XTI application programming interface, but it is not possible to utilize any of the STREAMS capabilities of an underlying Transport Provider Interface (TPI) stream. This approach is, unfortunately, also not ABI compliant to the SVID.

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. While being ABI compliant to SVID, under this approach it is still not possible to utilize any of the STREAMS capabilities of an underlying Transport Provider Interface (TPI) stream.

XTI over Sockets. Several implementations of XTI on BSD-style UNIX utilize the concept of XTI over Sockets (or 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. This approach is ABI compliant to the SVID and also allows the STREAMS capabilities of the resulting Transport Provider Interface (TPI) to be used (such as pushing and popping modules). This approach requires a rather complete STREAMS implementation, however, it does not require replacement of the BSD-style networking stack with an SVR4 style stack.

1.3 Standardization

It is interesting that both Sockets and STREAMS were implemented on the same operating system base (4.1BSD) at about the same time. Also, Dennis Ritchie implemented STREAMS and was a significant contributor to the initial BSD releases. BBN implemented the TCP/IP networking stack on 4.1aBSD using sockets, both released by the CSRG in 4.2BSD. At about the same time STREAMS went into System V Release 3 and the Network Services Utility (NSU) provided the first TLI implementations of TCP/IP networking for TLI. Perhaps it is not surprising that both interfaces provide similar functions:

4. Perhaps an exception is early versions of Digital UNIX.
5. This is the approach initially taken by Digital UNIX.
6. One was even available for Linux at one point.

TLI	Sockets
t_open()	socket()
t_bind()	bind()
t_listen()	listen()
t_connect()	connect()
t_rcvconnect()	select()
t_accept()	accept()
t_unbind()	bind()
t_close()	close()
t_snddis()	close()
t_sndrel()	shutdown()
t_sndreldata()	
t_rcvrel()	select()
t_rcvreldata()	
t_rcvuderr()	
t_snd()	send()
t_sndv()	writev()
t_sndudata()	sendto()
t_sndudata()	sendmsg()
t_rcv()	recv()
t_rcvv()	readv()
t_rcvudata()	recvfrom()
t_rcvudata()	recvmsg()
t_optmgmt()	getsockopt()
t_optmgmt()	setsockopt()

During the POSIX standarization process, networking and the Sockets interface were given special treatment to ensure that both the BSD and STREAMS networking approaches were compatible in the common application environment.[7] POSIX has standardized both the XTI and Sockets programmatic interfaces 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 release. The STREAMS networking provided by *Linux Fast-STREAMS* package provides POSIX compliant networking. Therefore, any application using a Socket or Stream in a POSIX compliant manner will be compatible with both BSD and STREAMS networking.

1.4 Linux

2 Objective

3 Description

3.1 STREAMS Networking

Figure 1 illustrates the organization of the classical STREAMS networking stack.

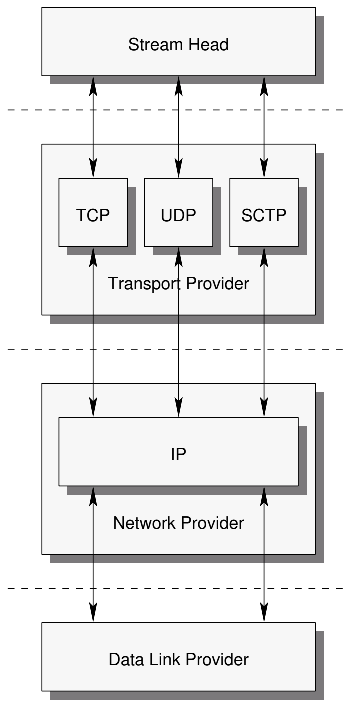

Figure 1: STREAMS Networking

User to Transport Interface. The interface between the Stream head and the uppermost module (transport) is a well-defined Transport Provider Interface (TPI). This is primarily a service primitive (message passing) interface.

Transport to Network Interface. The interface between the transport provider and the network provider is a well-defined Network Provider Interface (NPI). This is primarily a service primitive (message passing) interface.

Network to Data Link Interface. The interface between the network provider and the data link provider is a well-defined Data Link Provider Interface (DLPI). This is primarily a service primitive (message passing) interface.

3.2 BSD Networking

Figure 2 illustrates the organization of the classical BSD networking stack.

User to Transport Interface. The interface between the Socket layer and the uppermost (transport) protocol layer is a well-defined BSD socket-to-protocol interface. This is primarily a function pointer call interface.

Transport to Network Interface. The interface between protocol layers (transport and network) is a well-defined BSD protocol-to-protocol interface. This is primarily a function pointer call interface.

Network to Data Link Interface. The interface between the protocol layer and the device layer (network and interface) is a well-defined BSD protocol-to-iface interface. This is primarily a function pointer call interface.

7. For example, when a transport connection indication has been received with t_listen(3) the transport connection may have already been established before t_accept(3) or t_snddis(3) are issued. This permits t_listen(3) to be implemented using accept(3) and also permits the BSD networking stack to be used (a TPI implementation is capable of deferring accepting a connection at the protocol level and either accepting the connection (`T_CONN_RES`) or refusing the connection attempt (`T_DISCON_REQ`)). As another example, binding a socket with bind(3) to an address containing an address family of `AF_UNSPEC` has the same effect as t_unbind(3).

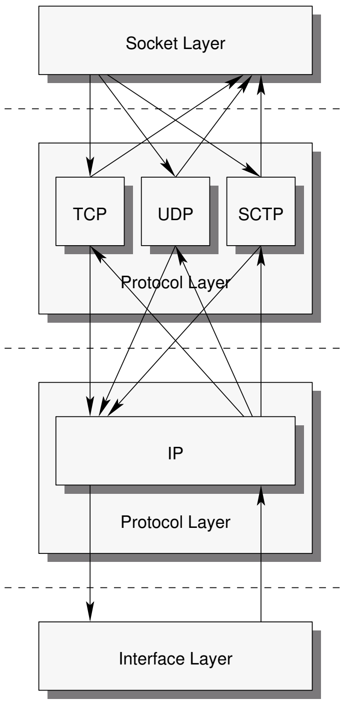

Figure 2: BSD Networking

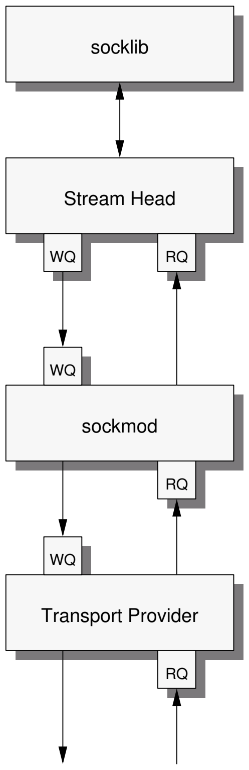

Figure 3: Socket Module

3.3 Hybrid Approaches

3.4 Linux Networking

3.5 Linux Fast-STREAMS Networking

4 Method

4.1 Compatibility Library

XTI Compatibility Library

This approach builds a library of socket functions that are implemented internally as calls to the XTI library. To maintain compatibilty with existing Linux sockets, the library distinguishes between XTI streams and native sockets. Underlying system calls (or input-output controls) are utilized on native sockets.

TPI Compatibility Library

This approach builds a library of socket functions that are implemented internally as service primitives passed to the TPI transport provider. To maintain compatibilty with existing Linux sockets, the library distinguishes between TPI streams and native sockets. Underlying system calls (or input-output controls) are utilized on native sockets.

4.2 Library and Cooperating Module

These approaches all invole pushing a module, named `sockmod` onto an open transport provider stream as illustrated in *Figure 3*.

Sockmod Approach 1

This approach pushes the `sockmod` STREAMS module onto the transport provider STREAM. A library of socket functions are implemented internally as calls to the cooperating module [VS90]. To maintain compatibilty with existing Linux sockets, the library distinguishes between Sockmod streams and native sockets. Underlying system calls (or input-output controls) are utilized on native sockets.

This is the approach of earlier Solaris releases [VS90].

Sockmod Approach 2

This approach pushes the `sockmod` STREAMS module onto the transport provider STREAM. A library of socket functions are implemented internally as calls to the cooperating module. Where this approach differs from the Sockmod approach above is that system calls are implemented directly as input-output controls issued to the socket module to emulate system calls. To maintain compatibilty with existing Linux sockets, the library does not need to distinguish between Sockmod streams and native sockets as both support the same set of underlying input-output controls (i.e. socksys input-output control calls available for iBCS compatiblity). Sockets opened in this fashion appear as STREAMS devices.

This is an intermediate approach that takes advantage of the socketsys input-output control calls available in the Linux kernel. The library can be made compatible with native sockets by passing the socket call to the normal glibc socket(3) function whenever the domain, type and protocol are not in the table.

4.3 Library and System Calls

These approaches all either transforms the inode associated with the Stream head into a socket, or attach a Stream head onto a socket. The objective in these approaches is to perhaps alter the socket(3) system call into a more specialized library call, but to keep all other socket system calls consistent with Linux and no need of replacing the other Linux socket system calls. This approach to system calls is problematic from the standpoint that both a `struct socket` and `struct sock` structure must be allocated. Linux socket calls expect both structures to be present and complete.

Perhaps the most straighforward way of doing this is to create both a `socket` and `sock` structure in the regular way for the Linux socket layer and provide a more specialized Stream head that is part of (or associated with) the `sock` structure.

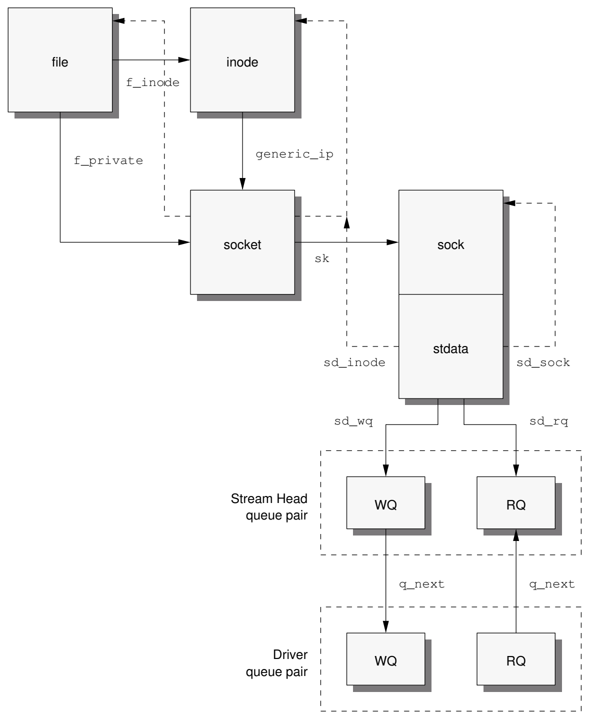

Figure 4: Socket Structures

Figure 4 illustrates the hybrid Stream head/Socket structure arrangement. Some of the particulars of the structures vary from kernel to kernel. For example, many kernels combine the `inode` and `socket` structures into a single structure. Some kernels will also combine the `sock` and `stdata` structures into a single structure. Other kernels separate these structures and provide pointers between the two. Also, *Linux Fast-STREAMS* currently allocates the `stdata` structure and its associated read and write queues as a single structure.

Sockmod Approach 3

This approach pushes the `sockmod` STREAMS module onto the transport provider stream. The `sockmod` module transforms the Stream head into a socket (from the viewpoint of the kernel), by establishing the arrangement shown in *Figure 4*.

A library implements the socket(3) system call as a library function to create sockets in this fashion. The library looks up the socket domain, type and protocol in a table and determines which transport provider device to open and then pushes the `sockmod` module onto the transport provider stream to transform it into a socket (from the viewpoint of the kernel). All other socket calls are performed as native sockets system calls.

This approach requires a thin library that implements the socket(3) system call. The library can be made compatible with native sockets by passing the socket call to the normal glibc socket(3) function whenever the domain, type and protocol are not in the table. This is the approach of later Solaris releases [Mar01].

Socksys Approach

This approach uses the soconfig(8) utility and sock2path(5) file (or the initsock(8) utility and the netconfig(5) file) to configure the system at boot time. The utility opens the `/dev/socksys` driver (or an anonymous device such as `/dev/ticlts`) and configures the system using input-output controls. Entries are added to an internal table containing socket domain, type, protocol and device path. The driver stores internally the identity of the transport provider (i.e. major device number).

The approach uses a thin library that re-implements the socket(3) system call. When the socket(3) system call is called, the library opens the `/dev/socksys` driver and issues an input-output control passing the arguments to the socket(3) system call (domain, type, protocol). The `/dev/socksys` driver creates a socket inode and a Stream head according to *Figure 4*, and attaches the driver of the type included in the configuration table. This STREAMS file is attached to an available file descriptor, which is returned from the input-output control. This file descriptor appears as a real socket to the reset of the system.

Another approach is to have the `/dev/socksys` driver detach its own driver queue pair and susbstitute the transport driver. The `/dev/socksys` driver also transforms it Stream head into a socket per *Figure 4* and returns its own file descriptor in the from the input-output control call.

All other socket calls are compatible with kernel system calls. If the domain, type and protocol are not present in the configuration table, the input-output control call can be passed to the sys_socket() call inside the kernel and a native socket of the requested type returned. The `/dev/socksys` driver itself implements the socket calls and converts them to TPI and other calls to the driver via the associated Stream head.

This approach requires only a thin library that implements the socket(3) system call and that is completely compatible with native sockets.

4.4 System Calls

This approach uses the soconfig(8) utility and sock2path(5) file (or the initsock(8) utility and the netconfig(5) file) to configure the system at boot time. The utility opens the `/dev/socksys` driver (or an anonymous device such as `/dev/ticlts`) and configures the system using input-output controls. Entries are added to a table containing socket domain, type, protocol and device path. The driver register a socket of the corresponding domain, type and protocol with the Linux socket system. To allow conflicting domains, if an existing domain is registered, a hack is used (bit added to the domain, e.g. `AF_INET | AF_HACK`). The driver stores internally the identity of the transport provider (i.e. major device number). When the Linux kernel sys_socket() system call is invoked, a socket structure is established and passed to the registered create function. The create function uses the socket domain, type and protocol to determine the STREAMS device to create. A Stream head is created and the transport provider driver attached in the same fashion as the open of a character device, except that the Stream head is attached to the socket inode. The `/dev/socksys` driver implements the remainder of the system calls.

This approach does not required a library: the existing C library socket functions are sufficient.

5 Results

6 Analysis

7 Conclusions

8 Future Work

9 Related Work

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.

[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.

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

[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/.