Portable and Performant Userspace SCTP Stack

Brad Penoff
Google, Inc.
Mountain View, CA, USA
Email: penoff@google.com

Alan Wagner
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada
Email: wagner@cs.ubc.ca

Michael Tüxen and Irene Rüngeler
Dept. of Electrical Engineering and Computer Science
Münster University of Applied Sciences
Steinfurt, Germany
Email: {tuexen,i.ruengeler}@fh-muenster.de

***Abstract*—One of only two new transport protocols introduced in the last 30 years is the Stream Control Transmission Protocol (SCTP). SCTP enables capabilities like additional throughput and fault tolerance for multihomed hosts. An SCTP implementation is included with the Linux kernel and another implementation called *sctplib* functions successfully in userspace on several platforms but unfortunately neither of these implementations have all of the latest features nor do they perform as well as the FreeBSD kernel implementation of SCTP. We were motivated to produce a portable implementation of the FreeBSD kernel SCTP stack that operates in userspace of any system because of both our desires to obtain a higher performance SCTP stack for Linux as well as to exploit recent developments in hardware virtualization and transport protocol onloading. Unlike any other userspace transport implementation for TCP or SCTP, our userspace SCTP stack simultaneously achieves similar throughput and latency as the Linux kernel TCP stack, without compromising on any of the transport's features as well as maintaining true portability across multiple operating systems and devices. We create a callback API and implement a threshold to control its usage; our userspace SCTP stack with these optimizations obtains higher throughput than the Linux kernel implementation of SCTP. We describe our userspace SCTP stack's design and demonstrate how it gives similar throughput and latency on Linux as the kernel TCP implementation, with the benefits of the new features of SCTP.**

I. Introduction

The exponential growth in the popularity of the Internet has seen more services like commerce and telephony move to IP-based networks. Many of these services require reliability so now more than 90% of Internet traffic use TCP as the reliable transport protocol. TCP did not provide the reliability and control necessary for the implementation of telephony over IP, so to overcome these deficiencies, the Stream Control Transmission Protocol (SCTP) was standardized [1], [2], [3]. It provides the user with more control over internal features and timers. SCTP also improves data integrity on the higher bandwidth interconnects of modern times by providing a stronger CRC32c checksum that avoids the false negatives that occur with the weaker 16-bit TCP checksum [4]. SCTP multi-streaming avoids head-of-line blocking amongst messages. Over multiple links, SCTP multi-homing allows for seamless fail-over in the event of network failure and with concurrent multipath transfer (CMT) [5], enables simultaneous data transfer over multiple links. All of these features have been available for SCTP for a decade where they have only partially appeared over TCP and UDP with efforts to enable multistreaming support at the application level [6], [7] as well as efforts for standardizing TCP to add multipath support [8] and a stronger CRC32c checksum [9].

Although the features of SCTP may be compelling, our initial tests could only match the performance of TCP using the FreeBSD kernel SCTP implementation. The FreeBSD implementation of SCTP has the best performance and is the most feature-rich SCTP implementation available, as it is maintained by the authors of the transport's RFCs in the IETF. Other implementations of SCTP have fallen behind and they are not as feature-rich nor are their performances as good.

The Linux SCTP stack [10] has been included in its default kernel distribution since 2.4.23 in November 2003 but it does not have all of the more recent SCTP RFCs nor has it been fully optimized; the performance of Linux SCTP does not perform as well as TCP. Siemens, the University of Essen, and Münster University of Applied Sciences have released an SCTP implementation called *sctplib* that runs in userspace but it also does not provide adequate performance or complete features [11]. The lack of availability of a high quality SCTP implementation on more popular platforms is an impediment to the wider spread adoption of SCTP.

Recently there are several developments that have made revisiting userspace stacks of interest. First, there is a renewed interest in protocol onload as protocol off-load devices arguably were a point-in-time solution that are less general [12]. There has also been more recent work starting to question whether transport protocols are too large to belong in the kernel. Van Jacobson made the argument that cache use can be improved with a user-level stack [13] on a multicore host because a user-level stack would ensure the transport processing is done on the same CPU as the application; this changes the "end" of the end-to-end principle common in network design from the host to the actual process or thread on a specific core, as was similarly proposed by Siemon [14]. Another development is that the newer generation of network adapters have started to have more support for network virtualization giving easier data paths to the NIC and providing protection domains on the NIC [15]. In addition, the latest NICs like the Intel 82599 also support an on-board CRC32c checksum computation, useful for iSCSI as well as SCTP, which eliminates a potential performance overhead posed by SCTP's stronger checksum. The combination of these advantages make it an opportune time to re-visit the question of how to make a fully-featured userspace SCTP stack.

The main contributions of our userspace SCTP stack are both the software provided to the open-source community as well as the optimizations described here that were able to simultaneously achieve portability on a variety of platforms and throughput and latency comparable to kernel transport implementations over Gigabit Ethernet. We have taken the feature-rich SCTP stack from FreeBSD and created the software components necessary to execute at user-level using standard techniques [16].[1] We introduce novel callback and threshold optimizations which result in a fully operational, device-agnostic stack running over either UDP or raw sockets that gives as good throughput and latency as kernel TCP on Linux but with the fault tolerance of SCTP multihoming and other benefits of SCTP. Our current implementation gives a very flexible implementation of a full-featured SCTP that can execute at the user-level on Linux, FreeBSD, Windows, Mac OS X and portable devices like the iPhone.[2] The core parts of our user-level SCTP stack still share the exact code as is compiled inside the FreeBSD kernel, thus we will be able to continue to take advantage of continued improvements and added features to the FreeBSD stack with our userspace design. Although there have been several simple implementations of transport protocols in simulation or in userspace for TCP [17], [18], [19], [20], none of these have had throughput and latency similar to the kernel implementation while still being full-featured, device-agnostic as well as portable across multiple operating systems, as what we have accomplished simultaneously for SCTP.

A major component of this work involved extracting the SCTP stack from the FreeBSD kernel to run in userspace. After giving an overview of related work in Section II, we introduce the initial design of our userspace stack in Section III. Following this, in Section IV we describe how we tuned our initial design in order to produce the best performance in a device-agnostic version of our userspace SCTP stack that matches kernel TCP with the additional features of SCTP such as the fault tolerance provided by SCTP multihoming. Finally, we conclude in Section V and offer ideas for future work.

II. Related Work

A. Other Userspace Stacks

Existing userspace stacks have not been able to achieve performance close to the kernel implementation while remaining feature-complete, as well as portable [19], [20], [18], [17], [11]. If performance is obtained, it is not portable. If it is portable, it is not performant. Our userspace stack provides all of the features available for SCTP yet it maintains high performance on different platforms.

Alpine [19] was a userspace TCP stack created in order to ease the development of network protocol development. They argue that by giving the developer the ability to make changes in userspace, they can shorten the development cycle of network protocol changes by enabling easier instrumentation and debugging. Many similar design issues are addressed in Alpine as in our work; these include the handling of sporadic control messages, their userspace implementation of timers, as well as how their networking stack interfaces to the upper and lower layers. Unlike our work, their design succeeds in coexistence with a kernel stack however they do not have performance as their primary focus; their bandwidth results are 25% that of the kernel network stack. Alpine's design was said to be portable in theory but it has only been tested on FreeBSD; a similar project called Daytona had nearly identical objectives as Alpine only for Linux [20].

Simultaneous to our work, the TCP stack has been recently extracted from the FreeBSD kernel [18]. This work is similar to ours in the sense that it comes from the same kernel so some of the times we had to extract services, e.g., socket structures, and other times we had to reimplement services, e.g., timers. However, it is different as well mostly because their focus is on functionality and not performance. They target lower bandwidth systems for use by virtualization, specifically 100 Mbps and wireless NICs. On the other hand, we target Gigabit NICs and low-latency environments. Our userspace stack achieves 23μs one-way latency while their work achieves 110μs one-way latency.

Solarflare [17] has created a full-featured, userspace TCP stack that completely bypasses the kernel. Their unique implementation achieves 15μs one-way latencies when used together with their specialized, Ethernet-compliant hardware [21]. While their userspace stack achieves lower latency than ours, ours maintains system portability since it still uses the drivers contained within the operating system; our approach can therefore run on several different platforms. To the authors' knowledge, no userspace SCTP implementation has been adapted to specific hardware to achieve $< 20\mu$s latency, however our work could serve as a foundation for future work to achieve this for SCTP as Solarflare has for TCP.

The *sctplib* is a user-level SCTP stack that executes on most major platforms [11]. Its focus has never been performance. The authors were able to apply lessons learned from developing and using this stack, we applied lessons learned to the userspace SCTP stack described in this paper, such as experiences with portability and interaction with the kernel. This stack is presently in bug maintenance mode and is not being actively developed for newer RFCs. Our starting point was the FreeBSD stack and our userspace stack has been integrated such that our stack can be built by changing the user directives and build parameters, so we therefore inherit bug fixes and new features added to the actively maintained FreeBSD kernel SCTP implementation.

B. SCTP Features Present over UDP or TCP

SCTP's new features provide applications with additional fault tolerance and therefore a potential for higher performance. Portions of these features have been added elsewhere at the application and transport layers. Some of the features

[1]The implementation work was done by Brad Penoff in collaboration with Humaira Kamal, Michael Tüxen, and Irene Rüngeler.

[2]Our SCTP stack is available at http://sctp.fh-muenster.de/sctp-user-land-stack.html .

are still in the standardization process and have yet to provide standard implementations.

SCTP multistreaming provides message ordering per stream which is also present over UDP in the Structured Stream Transport (SST) [7], but SST lacks other desired features like multihoming support. Similarly but in the domain of web browsing, Google's SPDY [6] is an application layer protocol who has shown 64% performance increases for web browsing through the use of batching. This specification uses a single TCP connection for funneling multiple HTTP requests which typically use a TCP connection per request, having to endure the costs of slow start. Increasingly large transfers over HTTP have resulted in applications demonstrating head-of-line blocking which SPDY avoids for TCP by implementing the multistreaming feature which originally motivated the advent of SCTP in telephony applications 12 years prior.

SCTP multihoming and the CMT extension allow endpoints with multiple network cards to fully utilize all available bandwidth for throughput and for additional fault tolerance. Given the usefulness of this feature, standardization efforts are underway in the IETF attempting to enable multipath capabilities in TCP [8]. Standard multipath for TCP has not appeared yet.

SCTP uses CRC32c as its checksum algorithm [22]. CRC32c used by the iSCSI storage standard as well for additional data integrity. TCP uses an additive 16-bit checksum which has been shown to accept one in 1×10^7 packets as valid despite being corrupted [4]. The CRC32c strengthens the message reliability by providing more protection by way of a stronger algorithm. The CRC32c has been proposed to add to TCP as well, but has not yet been accepted at the time this paper was written [9].

III. SCTP Userspace Stack Design

In order to create an SCTP stack that would perform better on Linux, a survey of potential starting points was conducted. The *sctplib* [11] already executes on Linux, however, for our purposes, it lacked the throughput and latency measurements that we wanted to achieve and also lacked the functionality that is present in the full implementation of the SCTP standard as it has lagged behind the RFCs. There are kernel versions of SCTP available but the version of SCTP with the best performance and that is the most feature-rich has been the FreeBSD stack that was originally developed by Randall Stewart, a co-inventor of SCTP. Linux has its own kernel implementation however it has not been optimized nor consistently maintained. The Java SDK supports SCTP but it utilizes the underlying kernel implementation on the host operating system [23]. The code used for the kernel FreeBSD SCTP stack has been used for both a Mac OS X [24] and Windows [25] version of the stack, which made it a good starting point for our work.

A representation of a kernel-based SCTP stack is shown above Figure 1-(1). An application uses an SCTP stack in the kernel by way of the Berkeley sockets API. Within the SCTP/IP stack, the transport protocol implementation forms a valid SCTP packet and passes it to the IP layer that then

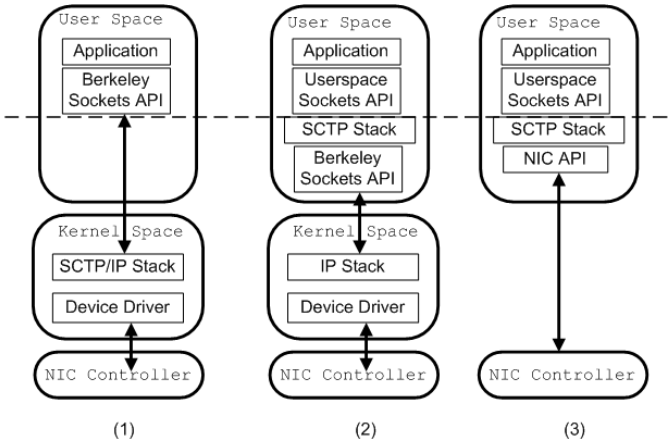

Fig. 1. SCTP Implementation Possibilities

performs routing table lookups for outward-bound packets; inward-bound packets are demultiplexed to the appropriate tuple and eventually that application's socket. Within the kernel, SCTP/IP interacts with the NIC controller by way of the device driver.

Under a kernel-based design, the host CPU processes the transport protocol in the operating system kernel, makes intermediate buffer copies, and also performs context switches between userspace and kernel space. As network speeds increase, these overheads cause an increased burden to the host CPU. Networking-related CPU overheads for a kernel-based TCP stack are measured to be 40% for transport protocol processing, 20% for intermediate buffer copies, and 40% for application context switching [26].

Additional copies can be avoided by using zero-copy between networking layers (e.g. TCP and IP) to bypass the kernel [27], as is shown in Figure 1-(3). This design avoids unnecessary context switches to/from kernel space because all operations are done in userspace or by the device. Copying is avoided by passing references in-between layers using either a slab allocator between software layers, or to hardware by way of a NIC API's zero-copy read/write functionality.

The major design challenge is to find general ways to reduce these overheads. Over the past years, several companies like NetEffect took advantage of the inability of the host OS to perform protocol processing for GigE and 10 GigE by TCP protocol offloading to the NIC card itself to achieve both zero-copy and kernel bypass. Typically, TCP offload devices are all-in-one solutions, although the Microsoft Chimney Architecture [28] has attempted to standardize the integration of TCP offload devices with the Windows operating systems. Nevertheless, transport offload solutions were more of a "point in time" solution [12] and more recently multicore and virtualization technology makes it easier to provide kernel bypass and more generic support on the NIC for protection and abilities such as Large Segment Offload (LSO), which can be used to achieve the performance gains of an offload device on the chip itself.

The basic goal to achieve the best performance was to attempt protocol onloading by moving the protocol stack to

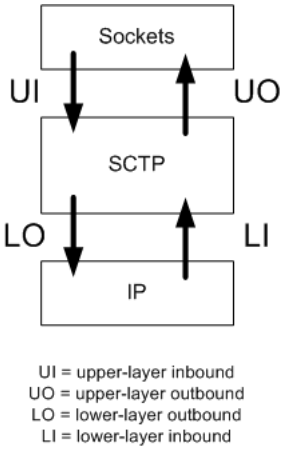

Fig. 2. SCTP Stack Interfaces and Directions

userspace, as is illustrated in Figure 1-(2). Moving only the transport protocol into userspace is intermediate to the overall final goal of kernel bypass; no one has done full kernel bypass for SCTP but it has been done in a device-specific manner for TCP [17].

There is merit in moving only the transport protocol into userspace, in addition to it being a path towards kernel bypass. Moving only the transport protocol into userspace makes the SCTP stack device-agnostic so it is more portable yet it is a good feature-rich implementation of SCTP at the user-level, as an alternative to the less tuned Linux SCTP stack as well as the *sctplib* userspace stack [11]; this also makes it possible to run SCTP on small, mobile devices such as cellular phones that only allow userspace-level development. Here we describe the design of our userspace stack that we use for protocol onloading, comparing it to how the same code works in the kernel. We first describe the lower-layer protocol (LLP) interactions then the upper-layer protocol (ULP) interactions, both pictured in Figure 2. Stack internal implementation issues for our userspace SCTP stack are then shared before we summarize our design.

A. LLP Interactions

As is shown beneath the SCTP stack in Figure 2, the SCTP implementation needs to interact with the layer below it. SCTP was originally specified over IP as its own transport as this was seen to be the architecturally correct solution by the SIGTRAN working group [3]. However UDP encapsulation [29] has been specified in case SCTP traffic needs to pass across legacy firewalls or if it needs to run on hosts that do not provide direct access to the IP layer such as the iPhone. Our userspace stack is capable of handling SCTP protocol data units (PDUs) layered over IP or UDP/IP, so all LLP interactions respectively occur using either a single raw IP socket that filters all SCTP traffic or a single UDP socket bound to a known tunneling port.

LLP Outbound – Throughout the stack, LLP outbound (LO) interactions pass SCTP PDUs downward towards the wire. When an application above SCTP sends data destined for some remote application, an LO interaction occurs. Elsewhere, the protocol generates LO interactions when it needs to pass some necessary control information. This happens when, for example, the protocol specification requires a Selective Acknowledgment (SACK) to be sent to acknowledge the receipt of data.[3] All LO interactions in the stack call an IP_OUTPUT macro which in our userspace stack, we implement as a simple `sendmsg()` call used with a raw socket for SCTP/IP or a UDP socket for UDP encapsulation. Therefore, like the kernel SCTP implementation, we presently make use of the kernel IP layer for routing as well as interfacing with the NIC.

LLP Inbound – When packets arrive from the wire and progress up into the SCTP stack in Figure 2, an LLP inbound (LI) interaction occurs. LI interactions occur unpredictably, however, it is important that the SCTP PDUs are handled promptly as the internal protocol state of the association is time-sensitive. In the kernel SCTP implementation, SCTP PDUs enter the SCTP stack by way of the `sctp_input()` method which is registered as a callback upon initialization of the network stack. This callback is fired when an SCTP packet arrives to the kernel IP implementation or to the assigned UDP encapsulation port.

In a userspace SCTP implementation, no such callback is registered as the kernel is operating within its own protection domain. Nonetheless, there is still a need to react responsively to LI interactions; in order to provide this, a thread is used to poll for the asynchronous arrival of SCTP PDUs. We use the portable `pthread` library to create a thread that polls with a blocking `recv()` on each lower-layer socket and passes the SCTP PDU into `sctp_input()`. We have one thread that filters SCTP/IP packets by way of the raw socket and in addition, we have another thread that filters UDP-encapsulated SCTP packets on our UDP socket.

The main difference between our userspace stack and the kernel stack is that LI interactions cross the kernel boundary in our userspace version whereas for the kernel, the LLP boundary is internal to the same protection domain inside the kernel. When inside the same protection domain, a callback mechanism can be used for asynchrony[30],[4] as is the case with the kernel SCTP implementation. On the other hand, for our userspace SCTP implementation, we make use of the kernel IP implementation which is in a different protection domain so we cannot use a callback mechanism for LI interactions to execute userspace code inside the kernel. Our thread calls `recv()` which uses Berkeley sockets thereby crossing into kernel space for the IP implementation. A wake-up occurs to traverse this kernel-userspace boundary to notify the blocking `recv()` call that an SCTP PDU has arrived and been placed in the lower-layer socket buffer.

[3]In SCTP, Selective Acknowledgements are the mandatory acknowledgement mechanism whereas in TCP, SACK is an alternative to cumulative acknowledgements. SCTP needs a more expressive acknowledgement scheme because data arrive out-of-order more commonly due to multihoming [3].

[4]We describe the design of our callback optimization for this userspace SCTP stack in Section IV-A.

B. ULP Interactions

At the upper layer, much of the FreeBSD kernel code that the userspace stack is based on assumes it is going to use FreeBSD's implementation of the Berkeley sockets API. Within the SCTP stack itself, the structures used to implement sockets in FreeBSD are intertwined throughout the code. Many different socket-related functions and structures are used extensively throughout the SCTP stack. In order for the SCTP stack to operate in userspace as shown in Figure 1-(2), these socket structures were exposed to userspace. We implement these socket structures and their related methods that are used by the SCTP stack itself.

In Figure 1, ULP interactions to the various SCTP stack implementations are represented by the dashed horizontal line. In the methods we exposed to userspace, these ULP interactions use locks to signal between the application and the transport stack. When an application no longer has to block on either a send or receive, a wake-up occurs via these locks from the stack to the user.

ULP interactions happen between the userspace stack and the application by way of our initial API which uses a `userspace_` prefix to the known Berkeley sockets API to denote the name of the methods we implemented with the semantic equivalent to their Berkeley socket counterparts. This custom socket API is listed in Table I. The disadvantage of using a different function name than the Berkeley socket API is that this requires applications to be ported to use our API directly. However, if applications were written using an API implemented in middleware like the Message Passing Interface (MPI), only the middleware implementation will have to change to use our API; the MPI programs themselves will retain their portability.

C. SCTP Implementation Internal

Inside the SCTP implementation itself, several other items needed to be implemented for the userspace implementation to operate on all platforms.

Memory Allocation – A transport protocol needs to avoid excessive copying and to quickly allocate/deallocate memory for SCTP PDUs. Inside of the kernel, PDUs are passed between the transport layer and below to the wire without an excessive number of copies; as the PDUs cross these layer boundaries, using an internal structure maintains a reference count and a pointer to the PDU's memory. In the Linux kernel, these memory management structures are called `sk_buff`s whereas in FreeBSD-based systems, they exist within the kernel known as `mbuf`s These structures are initially allocated in their respective kernel making use of a slab allocator to avoid fragmentation. This is done through its object caching strategy that is used when allocation and deallocation of memory of the same type/size is happening frequently, as is the case with `mbuf`s within a networking stack. Chunks stay in a per-CPU cache.

`mbuf`s are used throughout the FreeBSD SCTP kernel stack which our userspace SCTP implementation is based, so we re-implemented `mbuf`s and their support functions in userspace. Our stack can be configured to provide `mbuf` allocation using either `malloc()` from the heap or using the user-level libumem slab allocator [31], the latter of which is the default option for better performance on standard hosts whereas the former is used for smaller devices and for easier debugging.

Timers – As with any transport protocol, there are a number of timers needed to ensure reliable transmission as part of SCTP's state machine. A transport layer needs to keep track of time, deciding when to make responses or queries. An example of this is when establishing a connection. An INIT packet is sent and if the INIT-ACK is not received within a timeout, then the INIT needs to be resent. A more common example is a DATA chunk; if it is not acknowledged by a SACK before a timeout, this DATA chunk needs to be resent.

In our implementation, we provide the ability to schedule and deschedule timed transport interactions using a callout queue that runs in its own thread. This thread has an event loop and it maintains all timed events, firing the appropriate callbacks at their expiration times. If an event is not cancelled by another thread before that event timer expires, then it is serviced by the event loop when firing the associated callback at the desired time. This same approach is used in the FreeBSD kernel implementation.

The overhead of our timer implementation is minimal. Timeout values for a transport protocol are typically on the order of tens of milliseconds so we set our timer thread's event loop to only be awaken every 10 ms.[5] This is very coarse for a modern 2 GHz CPU to handle and therefore comes at a very low cost since it can handle 2.10×10^7 instructions elsewhere in 10 ms. When the thread awakes, it checks to see if it must deliver any expired timers for specific events. However, the majority of the time under standard operating conditions, there are no expired timers as their conditions have been met by packets received by the LLP thread. When conditions of a timeout event are met from any thread in the stack, their events are canceled and removed from the event loop, so even when the timer thread awakes, it typically has very little to do.

Platform Specific – Different operating systems have their own peculiarities. One difference is Linux puts the IP length into network byte order while other platforms do not. Another difference is in FreeBSD and Mac OS X, the length of the socket address structure, `sin_len`, is included in its length calculations while in Linux, it is not.

A final example of a difference across platforms is atomic operations. Because we have several threads operating on the same common data, we needed to add a locking solution. This was accomplished by creating an implementation for the ATOMICS macros used throughout the SCTP implementation. For Mac OS X, we made use of the `OSAtomic` interface where as for other platforms, we made use of the atomics built-in to GCC versions 4.1+ named `__sync_fetch_and_{subtract,add}`.

[5] A timer with a higher resolution could be used if there was a desire to set the timeout values to a smaller numbers.

Function	Description
`userspace_socket`	Returns a userspace socket.
`userspace_bind`	Binds a particular port and address set to a userspace socket.
`userspace_listen`	Enables server-side capabilities of a socket.
`userspace_accept`	Blocks until it returns a new userspace socket on a listening server.
`userspace_connect`	Connects to a remote userspace socket.
`userspace_sctp_sendmsg`	Sends data on a userspace socket.
`userspace_sctp_recvmsg`	Receives data from a userspace socket.
`sctp_setopt`	Allows the setting of socket options on a userspace socket.

TABLE I
SCTP USERSPACE CUSTOM SOCKET API FUNCTIONS

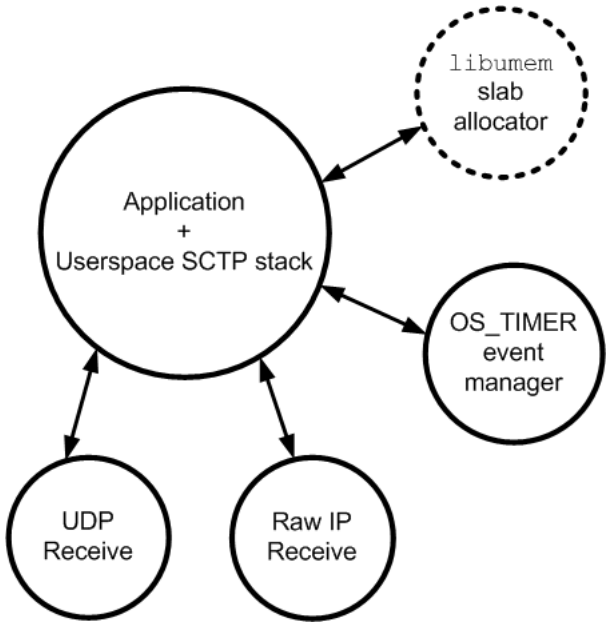

Fig. 3. Userspace SCTP Implementation Threads

D. Userspace Stack Project Properties

We now have a stack capable of running custom-built applications using our userspace SCTP stack. The SCTP portion of our stack runs entirely inside the same process as the application using it. State for application socket mappings or ports does not need to be communicated across the userspace-kernel boundary. Each application has their own instance of the complete stack, so there is no persistent state for default transport configuration settings. Changing most values for an application therefore requires a code change, however for convenience we read commonly used settings such as UDP encapsulation from an environment variable. Future designs could similarly store the values in a configuration file or on a persistent daemon.

Figure 3 summarizes the threads used in our userspace stack. There are potentially five threads in total used to implement our userspace SCTP, but potentially only four depending on the choice of memory allocator as libumem internally uses a thread and `malloc()/free()` do not. Under standard operating conditions, the timer event manager thread is inactive as well as is the receive thread that is not being used, e.g. the raw IP receive thread if we our application is running encapsulated over UDP.

Overall, this project provides a complete, full-featured userspace SCTP stack operating on Linux and other operating systems. Recently, this code has been contributed to the SCTP community by having committed to the common repository for the Windows, Mac OS X, and FreeBSD SCTP stack code. This SCTP userspace stack itself was a major contribution as a tool not only for use in this research but by the SCTP community as a whole. With our changes and use of preprocessor directives, we do not prevent the original code from working on their respective kernel platforms, so long as they are configured directly. Since it is put in this repository, going forward, our userspace stack continues to inherit future updates to the FreeBSD kernel stack.

The solutions presented here extracted the best kernel implementation of SCTP from FreeBSD and provided the necessary parts in order to make all of its features usable across all network devices and platforms. Our userspace SCTP stack is the first to implement CMT in userspace, a feature which provides the simultaneous transfer of data in environments with multiple network links. Techniques similar to those presented so far have been achieved by others for SCTP [11] and TCP [16], [17], [18], [19], [20], as Section II describes. However, we show next in Section IV how our novel optimizations achieve throughput and latency comparable to a kernel implementation.

IV. USERSPACE STACK THROUGHPUT, LATENCY, AND OPTIMIZATIONS

We describe and compare the performance of our userspace SCTP stack over a single Gigabit NIC configuration. We show its initial throughput results using a single-homed bandwidth test over Gigabit Ethernet. Our tests are conducted between two quad-core, dual socket (8 cores per node) Intel Xeon R X5550, 64-bit machines, running at 2.67 GHz. All machines have 12 GB of memory and two machines run Linux kernel 2.6.18-194.8.1.el5 while two ran the FreeBSD 9.0-CURRENT r214412 kernel.

This section describes our initial results together with the optimizations we designed and implemented to improve the throughput and latency. We first introduce in Section IV-A our callback API, an optimization that we designed for our userspace stack to increase throughput. After this in Section IV-B, we compare the results of the ported socket API to our callback API. Next in Section IV-C, we compare our userspace stack's throughput running on FreeBSD to Linux. After that

in Section IV-D, we show the benefit that adding a threshold can have in order to activate our send-side callback. Finally, in Section IV-E, we show the lower latency we can get when our userspace stack is no longer device-agnostic by showing that we can modify the device driver in order to achieve within 2μs of one-way latency compared to the Linux kernel TCP.

A. Callback API

In the kernel implementations, ULP interactions cross the kernel-userspace boundary, necessitating a wake-up operation to alert blocking operations that they are complete. For compatibility with the original kernel code, we implemented a ULP wake-up operation for the userspace stack, despite the fact that ULP interactions do not cross protection domains since both are in userspace. As a result, our initial userspace stack has an extra wake-up operation because in the userspace stack, there is a wake-up for the ULP interactions above the stack to the application as well beneath the stack to the socket being used for LLP interactions. Profiling confirmed that our initial implementation of ULP sockets spent an excessive amount of time acquiring locks.[6]

To avoid this extra wake-up within the socket implementation at the ULP boundary of our userspace stack, we designed an alternative API whose purpose was to bypass locking altogether since the application and transport stack were already in the same protection domain, namely userspace. We implemented a callback mechanism where all operations are non-blocking, so either the send or receive will complete immediately or the user can register a function that is fired as soon as the socket call is no longer blocking. For a receive, this callback is fired when the DATA chunk arrives. For a send, this callback is fired when sent data is acknowledged by the corresponding SACK packet, and space opens up in the send socket buffer beyond some specified threshold; this send-side threshold avoids wasting compute cycles by firing the send-side callback unnecessarily when a send is not going to succeed because a message is larger than the free space in the send socket buffer at the time. The threshold gives the transport protocol a hint to the application's next desired use, so it is therefore an example of a tighter integration of application and transport protocol; its benefits are shown in Section IV-D.

Recall from Figure 3 in the design description that there are receive threads, one for SCTP/IP packets and another for SCTP/UDP/IP packets. When one of these threads receives a packet, the packet is processed within that receive thread; in addition to doing the necessary processing required by SCTP, the registered callback function from the application is also executed in that same thread. This occurs without requiring a signal to the application as is the case when using costly locks contained within the socket structures. The callback mechanism is an efficient technique for using the userspace stack, compared to sockets. This approach is similar

[6]The exact results varied depending on the operating system of the test, as will be presented in Section IV-C.

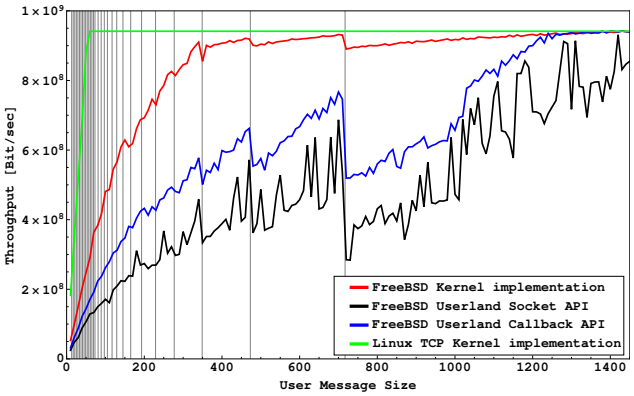

Fig. 4. Socket API versus Callback API for Userspace SCTP on FreeBSD

to APIs provided by Infiniband and other Ethernots[32]. It does not avoid locks altogether within the stack but it lessens their use at the ULP boundary; the LLP wake-ups are still necessary beneath our stack because we use the OS supplied implementation of Berkeley raw sockets and UDP sockets.[7]

B. Socket API versus Callback API

Our original userspace SCTP results running on FreeBSD are shown in Figure 4, with data points every 10 bytes. The faint vertical lines are the packing boundaries for multiple DATA chunks in one packet of MTU 1500, so naturally for SCTP which is message-based, there is a drop just after these boundaries when a packet cannot be efficiently packed.

The top two lines shown are the Linux kernel TCP curve as well as the FreeBSD kernel SCTP results to a bandwidth test. We repeat these two lines in Figures 4, 5, and 6 for reference. Although not shown in this figure, the FreeBSD kernel had the best throughput of all the SCTP implementations that we tried, matching the theoretical best for message sizes of 340 bytes and higher. Linux kernel TCP and FreeBSD kernel SCTP are equal for larger messages but as expected for bandwidth results, TCP can do better for smaller messages because it does not delimit the message boundaries in its stream of bytes. TCP's header is 20 bytes for all message sizes, while on the other hand, SCTP uses a 16 byte common header and then has a 12 byte header per DATA chunk. Small messages are CPU-bound for SCTP to do this extra processing therefore their throughput is less than TCP; the communication pipe can be filled more easily for SCTP as the message size grows and the CPU is no longer the bottleneck.

The lowest line in Figure 4 shows our bandwidth results from our initial API which is similar to the Berkeley sockets API, described in Section III. As one can see, this original userspace design, which uses the full userspace port of the FreeBSD socket structures, cannot achieve the bandwidth levels that the kernel implementations can.

[7]A callback mechanism like ours could avoid the costs of wake-ups in a kernel-based sockets implementation as well however, giving the application hooks to execute their code in the kernel can be unsafe as their execution would block the kernel from doing other more important tasks.

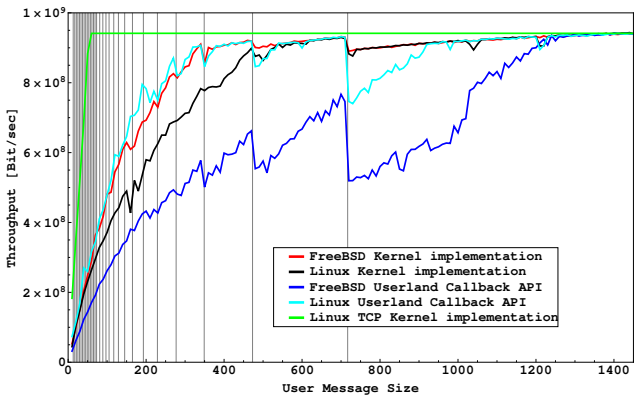

Fig. 5. FreeBSD versus Linux for Userspace SCTP with Callback API

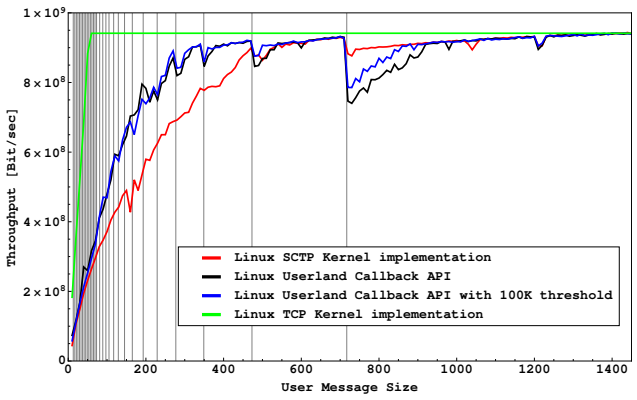

Fig. 6. Send-side Callback with each SACK versus at a 100KB Threshold

Figure 4 shows that the decrease in locking in the callback mechanism results in higher bandwidth benchmark results.[8] This figure shows that the lowest bandwidth results are of the original port that made use of the socket structures whereas the curve using our new callback mechanism that avoids the locks performed approximately 35% better than the socket API port. However, both of these userspace approaches when measured on FreeBSD are still worse than the kernel implementations. Further optimizations were necessary, as we continue to describe below.

C. Linux versus FreeBSD

In Figure 4, we had conducted our tests in FreeBSD. Using the same hardware, we tested our bandwidth results for Linux. These results are shown in Figure 5. Figure 5 shows several facts. To begin, the FreeBSD kernel SCTP implementation is better than the Linux kernel SCTP implementation, however both of these are better than the FreeBSD userspace results making use of the callback API reported previously. Figure 5 shows the userspace stack with the callback mechanism on Linux generally outperforms the kernel SCTP implementation in Linux, particularly for messages < 475 bytes. These results indicate that we could now achieve the desired throughput for an SCTP stack in Linux by using our portable userspace SCTP implementation.

In general, the results for our userspace stack were better in Linux than in FreeBSD. We decided to profile why this is the case. Using the `valgrind callgrind` tool which works with both FreeBSD and Linux, we found that the most time consuming functions are `pthread_mutex_unlock` and `pthread_mutex_lock`, where `pthread_mutex_unlock` is even called more often and needs more time. However, we saw that with Linux the impact of these functions was not as great (5% versus 17%), so it was logical to conclude that the implementations of `pthreads` are different on Linux and FreeBSD. It also indicated to us that by using Linux on the same hardware, we could obtain higher bandwidths with our userspace implementation as a result of the improvements to the `pthreads` implementation or using an alternative to `pthreads` such as coroutines that demonstrates consistent performance portability across platforms.

[8]The semantics of our callback mechanism are different than those provided by the socket API, however our bandwidth application could easily adapt from the socket API to the new semantics of our callback mechanism.

D. Callback Threshold

We also implemented a send-side callback, however, our bandwidth results still were not as high as a kernel implementation. To investigate, we performed bandwidth interoperability tests between our userspace stack and the FreeBSD kernel implementation, doing (1) kernel-send/userspace-receive followed by (2) userspace-send/kernel-receive. The results were uneven, indicating a bottleneck. The bandwidth for (1) was higher than the bandwidth for (2). The send-side implementation of our userspace stack was therefore the bottleneck. Not only is the send-side problematic because it has to deal with SACK processing, but the send-side callback was firing too frequently since our original send-side callback scheme invoked the callback with each SACK.

In light of excessive firing of the send-side callback, we modified our `register_send_cb` call so that the user can now choose to provide a threshold such that the send-side callback was only called when a user-specified amount of free socket buffer space is present. The idea was, that if the send-side callback is fired less often, we could improve the throughput. The threshold is compared to the free space in the socket send buffer. Typically this is the size of the next messages that will be sent. We compare our new approach with the original in Figure 6 and as is shown there, making use of a threshold increased the bandwidth for some message sizes up to 5%.

E. Driver Effects on Latency

Our initial latency tests showed that the latency of a kernel implementation was much faster than our userspace SCTP stack. We suspected that this was because the NIC's device driver coalesces interrupts by way of its automatic interrupt modification (AIM) scheme. Coalescing would increase message latency for small messages whereas for larger messages, bandwidth would be the limiting factor. In the default driver, AIM was enabled and there was no simple mechanism for disabling it other than recompiling the driver.

OS and Transport	Time (μs)
Kernel Linux TCP	21.48
Userspace FreeBSD SCTP CB w/o AIM	23.24
Userspace FreeBSD SCTP CB w/ AIM	25.2
Userspace Linux SCTP CB	30.5
Kernel Linux UDP	12.5
Kernel FreeBSD UDP w/o AIM	12.26
Kernel FreeBSD UDP w/ AIM	12

TABLE II
ONE-WAY LATENCY FOR 30 BYTE PAYLOAD

As a solution to this, a driver modification would be necessary. So far, all of our modifications have been either within the SCTP stack itself or adjusting the upper-layer boundary to the stack. We implemented a simple patch for the device driver to be able to adjust when AIM occurs. As shown in Table II for 30-byte messages, this slight modification to the FreeBSD driver can decrease the one-way latency of the userspace SCTP stack to be on par with that of the Linux kernel's TCP implementation. Without modifying the lower-layer of the userspace stack, we have the optimal combination of portability with high performance, as shown in Figure 1-(2). Future work involves integrating the lower-layer of the stack to a particular device as Solarflare [17] has done for TCP and is shown in Figure 1-(3); this will make our userspace stack less portable but will result in further decreases in latency than have been initially demonstrated here.

V. CONCLUSIONS

We have provided a tuned userspace implementation of SCTP which enables SCTP's additional features on most major platforms, including Linux. Our results show that having the transport protocol and the application in the same protection domain can increase performance, as suggested by Jacobson and Felderman[13]. By putting the transport stack into userspace and implementing our novel technique which supplies a callback triggered beyond a particular threshold, applications and the transport protocol have a tighter interaction. The optimizations to our userspace SCTP stack demonstrate higher throughput than the Linux kernel implementation of SCTP even without device-specific modifications. Making use of our callback mechanism, our SCTP userspace stack was able to perform on par with the Linux kernel TCP stack for large messages in bandwidth microbenchmarks. A simple driver adjustment decreased the latency measurements of our userspace SCTP stack for small messages by disabling interrupt coalescing. Future device-specific optimizations such as utilizing zero-copy to obtain full kernel bypass could yield further performance gains for the userspace stack.

These are the opinions of the authors, not their affiliations.

REFERENCES

[1] Stewart et al., "The Stream Control Transmission Protocol (SCTP)," Available from http://www.ietf.org/rfc/rfc2960.txt, October 2000.

[2] J. Yoakum and L. Ong, "An introduction to the Stream Control Transmission Protocol (SCTP)," Available from http://www.ietf.org/rfc/rfc3286.txt, May 2002.

[3] R. Stewart and Q. Xie, *Stream control transmission protocol (SCTP): a reference guide*. Addison-Wesley Longman Pub. Co., Inc., 2002.

[4] J. Stone and C. Partridge, "When the CRC and TCP checksum disagree," in *SIGCOMM*, 2000, pp. 309–319. [Online]. Available: citeseer.ist.psu.edu/stone00when.html

[5] J. Iyengar, P. Amer, and R. Stewart, "Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths," *IEEE/ACM Transactions on Networking*, vol. 14, no. 5, pp. 951–964, October 2006.

[6] Google, "SPDY project website," 2010, http://dev.chromium.org/spdy.

[7] B. Ford, "Structured streams: a new transport abstraction," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 4, pp. 361–372, 2007.

[8] IETF, "Multipath TCP (mptcp) Charter," 2011, available at http://datatracker.ietf.org/wg/mptcp/charter/ .

[9] A. Biswas, "Support for Stronger Error Detection Codes in TCP for Jumbo Frames, draft-ietf-tcpm-anumita-tcp-stronger-checksum-00 (work in progress)," *IETF*, May 2010.

[10] "Linux Kernel Stream Control Transmission Protocol (lksctp) project," available at http://lksctp.sourceforge.net/.

[11] M. Tüxen and T. Dreibholz, "The sctplib Userspace SCTP Implementation," 2009, available at http://www.sctp.de/sctp-download.html.

[12] S. Pope and D. Riddoch, "End of the Road for TCP Offload," Solarflare," Technical Report, April 2007.

[13] V. Jacobson and B. Felderman, "Network channels," Available from http://lwn.net/Articles/169961/, January 2006.

[14] D. L. Siemon, "The IP Per Process Model: Bringing End-to-end Network Connectivity to Applications," Master's thesis, University of Western Ontario, London, Ontario, Canada, 2007.

[15] Intel, "PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology," Available from http://download.intel.com/design/network/applnots/321211.pdf, 2008.

[16] R. A. Thekkath, T. D. Nguyen, E. Moy, E. D. Lazowska, and S. Member, "Implementing network protocols at user level," in *IEEE/ACM Transactions on Networking*, 1993, pp. 64–73.

[17] "Solarflare Communication (Ethernet Accelerator)," Available at http://www.solarflare.com.

[18] A. Kantee, "Environmental Independence: BSD Kernel TCP/IP in Userspace," in *Proceedings of AsiaBSDCon 2009*, 2009.

[19] D. Ely, S. Savage, and D. Wetherall, "Alpine: a user-level infrastructure for network protocol development," in *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*. Berkeley, CA, USA: USENIX Association, 2001, pp. 15–15.

[20] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum, "Daytona: A User-Level TCP Stack," Available from http://nms.csail.mit.edu/ kandula/data/daytona.pdf.

[21] J. Layton, "Cluster Interconnects: The Whole Shebang," http://www.clustermonkey.net/content/view/124/33/, April 2006.

[22] J. Stone, R. Stewart, and D. Otis, "Stream Control Transmission Protocol (SCTP) Checksum Change," Available from http://www.ietf.org/rfc/rfc3309.txt, September 2002.

[23] Chris Hegarty, "SCTP in Java," available at http://openjdk.java.net/projects/sctp/.

[24] M. Tüxen, "An SCTP network kernel extension for Mac OS X," 2010, available at http://sctp.fh-muenster.de/sctp-nke.html.

[25] B. Cran, "SctpDrv: an SCTP driver for Microsoft Windows," 2010, available at http://www.bluestop.org/SctpDrv/.

[26] B. Hausauer, "iWARP: Reducing Ethernet Overhead in Data Center Designs," Available at http://www.eetimes.com/design/communications-design/4009333/iWARP-Reducing-Ethernet-Overhead-in-Data-Center-Designs, November 2004.

[27] J. Chase, A. Gallatin, and K. Yocum, "End-System Optimizations for High-Speed TCP," *IEEE Communications Magazine*, vol. 39, pp. 68–74, 2000.

[28] S. Jang, "Microsoft Chimney: The Answer to TOE Explosion?" Available at http://margallacomm.com/downloads/TOE_Chimney.pdf, 2008.

[29] M. Tuexen and R. Stewart, "UDP Encapsulation of SCTP Packets," Available at http://tools.ietf.org/html/draft-tuexen-sctp-udp-encaps-05, July 2010.

[30] R. Gopalakrishnan and G. M. Parulkar, "Efficient user-space protocol implementations with qos guarantees using real-time upcalls," *IEEE/ACM Trans. Netw.*, vol. 6, pp. 374–388, August 1998.

[31] Omni-TI-Labs, "Libumem project website," 2010, available at https://labs.omniti.com/trac/portableumem/.

[32] P. Balaji, W. Feng, and D. K. Panda, "Bridging the Ethernet-Ethernot Performance Gap," *IEEE Micro*, vol. 26, pp. 24–40, 2006.