Linux-Integrated TCP Acceleration as a Service

Amanda Austin
University of Texas at Austin

Henrique Fingler
University of Texas at Austin

Timothy Stamler
University of Texas at Austin

ABSTRACT

Netowkring speeds have increased while CPU speeds have not. As a result, an increasing portion of packet processing time is spent in the kernel networking stack. To mitigate this effect, TCP Acceleration as a Service (TAS) splits TCP processing into a fast path and a slow path, both of which operate as userspace processes. In doing so, however, TAS loses some information about the network when compared to an in-kernel networking stack (e.g., firewalls, ARP tables). We present a Linux-integrated TAS that interfaces with Linux via a virtual network device. TAS duplicates some slow path packets to Linux via this virtual device, observes Linux's response, and then mimics that response. We show that this method can allow TAS to use Linux's information about the network while retaining the performance of TAS fast path operations.

ACM Reference Format:
Amanda Austin, Henrique Fingler, and Timothy Stamler. 2018. Linux-Integrated TCP Acceleration as a Service. In *Proceedings of CS395T: Datacenters.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Networking speeds have become faster while CPUs have not, causing network packet processing efficiency to become important for datacenter networks. Datacenter applications continue to want high throughput and low latency access to the network along with the guarantees provided by TCP: lossless in-order delivery of packets, but this comes at the cost of consuming an increasing fraction of CPU processing resources. For example, nearly 70% of packet processing time for a simple echo server application is spent in the Linux networking stack [5].

To cope with this, many alternative TCP stacks have been proposed that seek to increase the efficiency of packet processing. TAS (TCP Acceleration as a Service) splits TCP packet processing into a fast path and a slow path. The fast path handles common data path operations such as handling in-order delivery of packets from established connections and generating acknowledgements. The slow path handles less common, control path operations such as connection management, congestion control, and connection timeouts. Both the fast path and slow path operate as user-level processes.

Implementing a TCP stack in userspace comes with a few drawbacks. Namely, the Linux TCP stack contains a lot of functionality and information about the network that is hard to replicate in userspace. Ideally, a userspace networking stack should make the same decisions about connection management, security, congestion, etc. as the Linux stack.

We present Linux-Integrated TCP Acceleration as a Service, an extension to TAS that interfaces with Linux for some slow path operations. The slow path now sends some packets to Linux, observes Linux's response, and mimics it. In this way, TAS can gain some of the information and functionality of the Linux TCP stack, such as firewall and network information (e.g., ARP tables), while retaining the performance of fast path operations.

Our paper makes the following contributions:

- We design and implement a method for the TAS to interface with Linux for some slow path operations (connection setup and teardown, ARP) in order to gain information and functionality.
- We evaluate our implementation and show that we introduce no overheads for fast path operations. Connection setup and ARP slow down significantly, but these operations are uncommon enough that they do not affect the throughput seen by the application.

In the remainder of our paper, we provide some background on TAS and virtual network devices in Section 2. We discuss the design and implementation of Linux-integrated TAS in Section 3. We evaluate our implementation in Section 5 and finally conclude and discuss future work in Section 6.

2 BACKGROUND

2.1 TCP acceleration as a service (TAS)

It is known that operating systems cause a large overhead on the network stack. Arrakis [5] showed that if we remove the kernel from the packet processing path and do everything from user-level, the time required to process one packet can be up to 16 times less.

Because of this and network speeds ramping up, there has been plenty of work implementing user-level network stacks [2–4, 6] with different ideas that try make general adoption easier.

One of these ideas is TCP acceleration as a service (TAS or SplitTCP ****TBD: check naming on intro/abstract***). The idea behind TAS is that the TCP stack can be divided into two parts: a slow path, which does the connection setup and teardown, and a fast path, where the data packets are sent and/or received. Figure 1 shows the general idea of TAS; the user's application use an unmodified POSIX API to talk to a user-level TCP library. The fast path can directly read from and write to the network card through DPDK [1]; when data packets arrive or need to be sent out, the fast path directly communicates with the user-level library; when it gets packets that have to be handled by the slow path (such as SYN packets), the packet is offloaded to it through shared memory. Because the two paths are executed in different threads, the slow path doesn't affect the performance of the fast path.

CS395T: Datacenters, Fall 2018, University of Texas at Austin

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

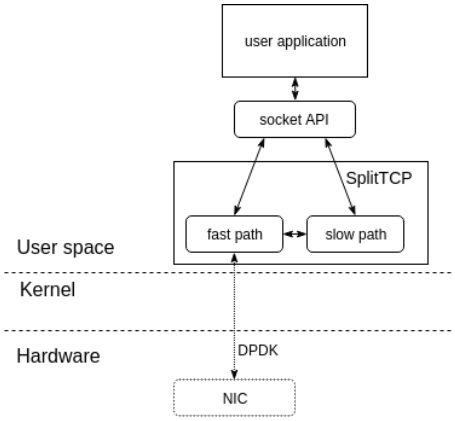

Figure 1: The idea behind TAS is split network operations between a slow and a fast path, both in user level, completely avoiding the kernel.

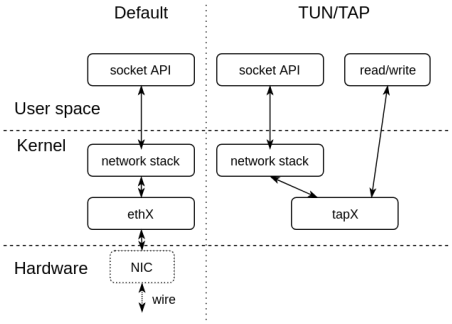

Figure 2: The difference between the usage of a regular NIC from a TUN/TAP device. In the latter the data is not sent to the outside, but stored as raw data in a kernel space buffer which an application can read and write to.

This idea yields very good performance for the average case of a datacenter: connections are long-lived and set up pretty infrequently; thus data packets are predominant, making the fast path much more active than the slow path. One issue with this solution (and most other user-level stacks) is that all the desired kernel functionality are lost or need to be reimplemented, such as packet filtering, firewalls, control, etc.

2.2 TAP devices

One of the tools used in this work was TUN/TAP devices. These are virtual network devices that are purely software (hence the "virtual"). These devices mimic a physical NIC, the only difference is that instead of receiving/sending data to the outside through a wire/radio waves, it is done so to a buffer in kernel space that an application can read/write through the POSIX file API. Figure 2 has a basic representation of both concepts: when using a regular (wired) NIC, the kernel gets calls from user space, and sends/receives from a physical device. When using a TUN/TAP device, the data is read/written to a buffer in kernel, and an application in user space can capture and write raw network data to and from it, emulating a real outside connection.

The different between TUN and TAP is the network layer in which they work. TUN devices work in layer 3 (IP packets), while TAP devices work in layer 2 (ethernet frames).

3 DESIGN

3.1 SplitTCP Design

SplitTCP seeks to address a number of problems in the realm of datacenter packet processing such as effciency, predictability, and workload proportionality. Several important design decisions were made to accomplish these, but we will focus on just two of those: the split into a fast path and a slow path, and the implementation of a userspace TCP stack.

One key way that SplitTCP achieves high connection scalability and predictability is by dividing TCP funcationality into two components: a fast path and a slow path. The fast path handles common-case packets and detects exceptions that need to be handled in the slow path. The main functions that the fast path handles are common case send/receive, where the fast path directly reads or writes packet payloads from application buffers and interacts directly with the network, fast ACK handling, where the fast path will either generate or consume ACK packets quickly without having to invoke the slow path, and efficient exception recognition and forwarding.

In doing this, heavyweight TCP operations are taken out of the data path and relegated to the slow path, where they can be handled without affecting the performance of other flows in the fast path. The slow path does costly, infrequent operations like connection setup and teardown, timeouts, and enforcing congestion control. The slow path also handles out of order packets, but these are very infrequent in a datacenter environment.

In order for this separation and the individual components of the fast path and slow path to stay efficient, everything is implemented in a userspace TCP stack. This provides all the normal functionality of TCP without having to switch back and forth between user and kernel space. This is good for performance and ease of programming, but isolates the TCP stack from Linux. While we don't want to rely on Linux for performance operations, it does contain information about the network and a configurability that would be useful in a datacenter environment.

3.2 Integration Goals

To most effectively interface with Linux, we take advantage of the previously described SplitTCP design choices in order to minimize our impact on common case packet processing.

First, we don't make any modifications to the fast path code or data structures. We prioritize making changes around the fast path, but add no additional lines of code to common case packet processing, and no additions and minimal interaction with fast path data structures to avoid incurring contention, cache problems, or race conditions. By following this rule, we shouldn't incur any direct overheads to the cases that we care most about.

Secondly, we don't worry about performance in the slow path as long as it doesn't incur any scheduling or blocking problems with the fast path. We have very little control over how often Linux

will deliver packets to the slow path, so we want to make sure this doesn't negatively affect the fast path threads when they might be co-scheduled with the slow path.

3.3 Compromises Made

In order to accomplish these goals, we made some compromises to certain problems in our implementation. In these cases, we could have used few lines of code or perhaps gotten better performance for the slow path operation, but at the expense of potentially slowing down the fast path.

First, as previously discussed, the fast path naturally consumes all ACK packet, including during connection setup. In order to most faithfully setup the connection, we would need to modify the fast path to either forward ACK packets to the slow path, or write packets directly to the tap device. Instead of absorbing these costs in the fast path, we instead manually create a new ACK packet during connection setup in the slow path and write it to the tap device.

Second, we maintain separate sequence numbers between Linux and SplitTCP and between SplitTCP and remote hosts. SplitTCP immediately initializes fast path state upon receiving a SYN packet and doesn't update it after that unless there's some kind of exception. Although this takes place in the slow path, this would require adding some kind of complexity, as we won't know the sequence number Linux wants to use until it provides a SYNACK. We can address this by doing one of the following: adding an artificial delay or some kind of signal from the thread interacting with Linux in the middle of SYN processing to make sure the sequence number is available when fast path state is initialized, initializing fast path state at a later time, modifying the fast path to identify either outgoing SYNACK packets or incorrect outgoing sequence numbers and adapting, or keeping separate sequence numbers. The final solution ended up being the simplest; the first will slow down connection setup even more, the second may create additional race conditions or problems in connection setup, and the last will slow down the fast path in some way.

Third, similarly to the first compromise, we handle ARP packets manually in the slow path, as they are not always forwarded to the slow path. By default, the fast path only forwards ARP packets to the slow path if it is an IP address that it doesn't have state for. As connection state is initialized immediately upon a SYN packet being received, if we try to forward any Linux ARPs to the network for that connection, the replies will be dropped by the fast path. In this case, instead of forwarding all ARP packets to the slow path or delaying initializing connection state again, we fake ARP responses to Linux requests in the slow path by searching SplitTCP's ARP table and manually crafting packets.

4 IMPLEMENTATION

In order to achieve a Linux-integrated SplitTCP implementation, the following changes were implemented and functionalities added.

4.1 Tap Initialization and Interaction

At the time of SplitTCP's initialization, a tap device is created with Linux. We control the name and IP of this tap device, but Linux and anyone using the operating system with root privileges can configure it to do whatever they want, and SplitTCP will reflect these changes. At initialization, we need to save the MAC address that is assigned to the tap device so that future packets written to the tap device can copy and use it.

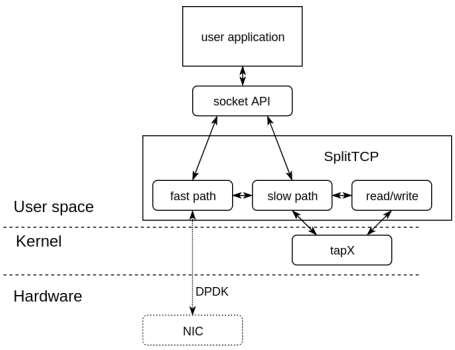

Figure 3: Design of TAS with kernel integration. TAS now replicates all slow path operations to the kernel, checks what it did by reading raw packets from the TAP device and mimics its decisions.

Writing packets to the tap device is simple and can be done from anywhere in the slow path. It is merely a function call with the buffer and a length, but we make sure to copy the saved MAC address for the tap device over the desination MAC field in any packets being written to the tap device.

Reading packets from the tap device is done in a separate thread that can block with epoll while waiting for packets to be received from Linux. Once packets are received, they are parsed and handled according to headers, flags, and existing connection state. With this parsing already done, it enables us to more easily handle more types of Linux packets and add more functionality.

4.2 Tap Control Packets and Network Interface

To more easily create and format packets to write to the tap device, a new interface was written based on the existing SplitTCP control packet creation interface, but reversed. When a control packet such as an ACK needs to be generated, connection state is looked up and crafted into a packet, with SplitTCP being the source and the remote host being the destination. Instead, with our tap control interface, a packet is crafted with the remote host as the source and Linux as the destination. This reversal applies to IP addresses, MAC addresses, sequence and acknoledgement numbers, and TCP timestamp options.

In addition to a new interface for generating packets for the tap device, a new interface was needed for forwarding raw packets from the tap device to the network. By default, SplitTCP only has the existing control interface for creating and sending certain packets to the network from the slow path, but not an interface for sending raw buffers directly to the network. We added this functionality so that we can forward packets directly from Linux to the remote host with minimal modifications.

4.3 Modified Socket Library

Because Linux does not just acknowledge arbitrary SYN packets, we had to do some extra work at the library level in order to prepare Linux. Typically, the application would interact with Linux through the socket library, but with SplitTCP, all socket operations are intercepted and forwarded instead of being delivered to Linux. In this environment, Linux is completely unaware of the application listening for connections. To rectify this, we duplicate all necessary socket operations through libc. When the application calls a socket operation that we need, the interposition layer on top of libc will call both SplitTCP and libc so that Linux is aware of all the same information that SplitTCP is. The functions we currently handle for this are socket, bind, fcntl, setsockopt, listen, and accept.

To make sure that file descriptors aren't getting mixed up, we maintain a map of SplitTCP file descriptors to Linux file descriptors and only return SplitTCP descriptors to the application. Whenever one of the aforementioned functions is called, the SplitTCP file descriptor provided by the application is cross-referenced with the map in order to find the proper file descriptor to use with libc.

4.4 ARP Support

In our implementation of SplitTCP, Linux cannot interact directly with remote hosts, and ARP requests and replies are no longer handled after connection state is initialized. Because Linux will likely need to do an ARP request to obtain the remote host's MAC address, and because we don't want to make any modifications to the fast path, we handle ARP requests in the slow path.

ARP replies are generally sent by the host to which the corresponding address belongs, but in order to provide Linux the information it needs we fake these replies inside the slow path. Upon receiving an ARP request from Linux, we search through SplitTCP's ARP table to see if the address is there. If it is, we generate a reply and write it back to Linux. If it is not, we forward the packet to the network. Because SplitTCP was unaware of this address, we know it does not maintain any connection state for it, and the corresponding ARP reply should be forwarded to the slow path, where it can be written to the tap device.

4.5 Connection Setup

Finally, we will outline the general changes we had to make to SplitTCP to get it to cooperate with Linux during connection setup. Once the SYN packet arrives, it will naturally be forwarded to the slow path for connection establishment. After allowing SplitTCP to do its normal connection state initialization, we write the SYN packet to Linux. At this point, SplitTCP would generate an event for the SYNACK to be generated asynchronously. We remove this funcationality and instead wait for Linux to send a SYNACK. Linux will first do an ARP request, which we handle, followed by a SYNACK. We forward this SYNACK to the network, then do the normal SplitTCP work of updating the connection state and signalling the application, as well as the additional work of generating an ACK packet to send to Linux.

5 EVALUATION

In this section, we evaluate our implementation of Linux-Integrated TAS. Our evalutation seeks to answer the following questions:

	Latency (us)					
	Original TAS			Linux-Integrated TAS		
# Connections	Avg	99%	99.99%	Avg	99%	99.99%
1	60	70	79	61	62	78
2	63	65	97	62	64	82
4	65	68	105	65	89	118
8	71	79	147	70	78	151
16	68	77	144	69	76	146
32	72	97	170	74	104	153

Table 1: Average and tail latency of RPC Echo server microbenchmark

- Do our changes affect the performance of the fast path?
- What is the performance of the slow path?

5.1 Evaluation Setup

To answer the above questions, we run a simple RPC echo server microbenchmark. A client sends a packet with a 64 byte payload to a server, which echos the packet back to the client. Both client and server are single threaded. The server machine is an Intel Xeon Gold 6138 system at 2.0 GHz with a 1G NIC. The client machine is an Intel Xeon E3-1225 v3 system at 3.3 GHz with a 1G NIC. Both client and server run Linux kernel 4.18.

5.2 Fast Path Performance

Table 1 shows the average and tail latencies of the RPC echo microbenchmark with TAS and Linux-Integrated TAS. The average-case latency of the fast path of Linux-Integrated TAS is within 3% of TAS. The performance at the tail varies a bit more between the two versions, but is within 25% of TAS in the worst case.

Figure 4 shows the throughput of the RPC echo server microbenchmark using the original TAS and Linux-Integrated TAS. The throughput of both versions are very similar.

Discussion. Our evaluation results show that the fast path performance of Linux-Integrated TAS is very similar to the fast path performance of the original TAS. This result aligns with the fact that we made no changes to the fast path. Additionally, slow path operations such as connection setup and teardown happen infrequently enough that they do not affect the performance of the fast path.

5.3 Slow Path Performance

Our performance results show that the slow path performance is reduced drastically compared to the original TAS. Connection setup times were reported on the order of seconds. This is due to the fact that we now incur system call overheads on the slow path. We must wait for Linux to handle the packets we send to it before we can observe its response. For example, we must issue a blocking accept call to Linux to ensure that TAS does not prematurely move on to the next connection before observing Linux's response.

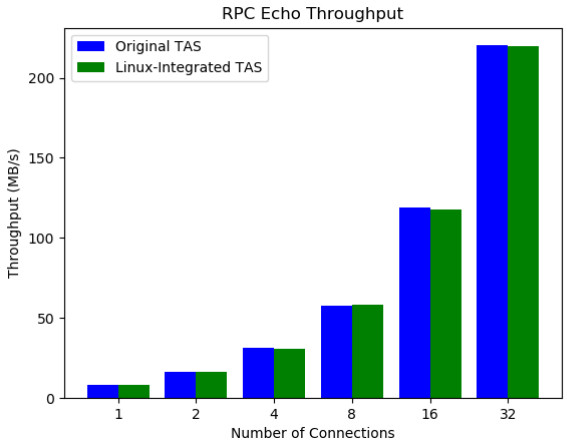

Figure 4: RPC echo throughput for a single threaded client and server.

6 CONCLUSION

Integrating kernel functionality into TAS did not affect the fast path operations, which is where it gets most of its performance benefits. The slow path took a hit in performance due to proxying system calls on every slow path operation and having to read/write from a TAP device, which incurs plenty of mode switches. Because slow path operations are infrequent, we think that the tradeoff of slower connection setup and teardown is worth it to add powerful kernel network functionality such as packet filtering without having to reimplement it all in user space.

REFERENCES

[1] [n. d.]. DPDK: the Data Plane Development Kit. https://www.dpdk.org/. Accessed: 12-09-2018.

[2] Thorsten Von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. 1995. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *In Fifteenth ACM Symposium on Operating System Principles*.

[3] David Ely, Stefan Savage, and David Wetherall. 2001. Alpine: A User-level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd Conference on USENIX Symposium on Internet Technologies and Systems - Volume 3 (USITS'01)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1251440.1251455

[4] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[5] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. 33, 4, Article 11 (Nov. 2015), 30 pages. https://doi.org/10.1145/2812806

[6] C. Zheng, Q. Tang, Q. Lu, J. Li, Z. Zhou, and Q. Liu. 2018. Janus: A User-Level TCP Stack for Processing 40 Million Concurrent TCP Connections. In *2018 IEEE International Conference on Communications (ICC)*. 1–7. https://doi.org/10.1109/ICC.2018.8422993