Rekindling Network Protocol Innovation with User-Level Stacks

Michio Honda*, Felipe Huici*, Costin Raiciu†, Joao Araujo‡, Luigi Rizzo§
NEC Europe Ltd*, Universitatea Politehnica Bucuresti†, University College London‡, University of Pisa§
{first.last}@neclab.eu, costin.raiciu@cs.pub.ro, j.araujo@ucl.ac.uk, rizzo@iet.unipi.it

ABSTRACT

Recent studies show that more than 86% of Internet paths allow well-designed TCP extensions, meaning that it is still possible to deploy transport layer improvements despite the existence of middleboxes in the network. Hence, the blame for the slow evolution of protocols (with extensions taking many years to become widely used) should be placed on end systems.

In this paper, we revisit the case for moving protocols stacks up into user space in order to ease the deployment of new protocols, extensions, or performance optimizations. We present MultiStack, operating system support for user-level protocol stacks. MultiStack runs within commodity operating systems, can concurrently host a large number of isolated stacks, has a fall-back path to the legacy host stack, and is able to process packets at rates of 10Gb/s.

We validate our design by showing that our mux/demux layer can validate and switch packets at line rate (up to 14.88 Mpps) on a 10 Gbit port using 1-2 cores, and that a proof-of-concept HTTP server running over a basic userspace TCP outperforms by 18–90% both the same server and nginx running over the kernel's stack.

1. INTRODUCTION

The TCP/IP protocol suite has been mostly implemented in the operating system kernel since the inception of UNIX to ensure performance, security and isolation between user processes. Over time, new protocols and features have appeared (e.g., SCTP, DCCP, MPTCP, improved versions of TCP), many of which have become part of mainstream OSes and distributions. Fortunately, the Internet is still able to accommodate the evolution of protocols: a recent study [10] has shown that as many as 86% of Internet paths still allow TCP extensions despite the existence of a large number of middleboxes.

However, the availability of a feature does not imply widespread, timely deployment. Being part of the kernel, new protocols/extensions have system-wide impact, and are typically enabled or installed during OS upgrades. These happen infrequently not only because of slow release cycles, but also due to their cost and potential disruption to existing setups. If protocol stacks were embedded into applications, they could be updated on a case-by-case basis, and deployment would be a lot more timely.

For example, Mac OS, Windows XP and FreeBSD still use a traditional Additive Increase Multiplicative Decrease (AIMD) algorithm for TCP congestion control, while Linux and Windows Vista (and later) use newer algorithms that achieve better bandwidth utilization and mitigate RTT unfairness [22, 26]. From a user's point of view there is no reason not to adopt such new algorithms, but they do not because it can only be done via OS upgrades that are often costly or unavailable. Even if they are available, OS default settings that disable such extensions or modifications can further hinder timely deployment.

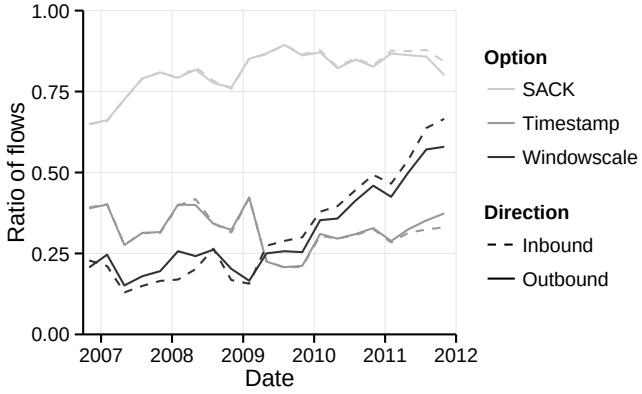


Figure 1: TCP options deployment over time.

Figure 1 shows another example, the usage of the three most pervasive TCP extensions: Window Scale (WS) [12], Timestamps (TS) [12] and Selective Acknowledgment (SACK) [16]*. For example, despite WS and TS being available since Windows 2000 and on by default since Windows Vista in 2006, as late as 2012 more than 30% and 70% of flows still did not negotiate these options (respectively), showing that it can take a long time to actually upgrade or change OSes and thus the network stacks in their kernels. We see wider deployment for SACK in 2007 (70%) compared to the other options thanks to it being on by default since Windows 2000, but even with this, 20% of flows still did not use this option as late as 2011. The argument remains unchanged for the Linux kernel, which had SACK and TS on by default since 1999 and WS since 2004.

These are not problems of the past. Something similar will happen in the future with FastOpen [20], which permits the exchange of data in SYN packets. Likewise for Multipath TCP [21], which can improve overall throughput by distributing packets over distinct end-to-end paths [29]. Besides slow deployment, other problems with in-kernel protocol stacks include difficulty of development and low portability, which also hamper maintenance and code reuse.

*We used a set of daily traces from the WIDE backbone network which provides connectivity to universities and research institutes in Japan [3].

Moving all transport layer functionality to user-space on top of in-kernel UDP as in QUIC [28] is not a panacea: the widespread adoption of middleboxes in today's networks means that UDP does not always get through certain paths and that even TCP often struggles [10], making the ability to easily modify TCP, and more generally the transport layer, paramount.

To tackle these issues, we argue for moving network protocol stacks up to user-space in order to accelerate innovation and feature deployment. User-level protocol stacks have been proposed several times in the 90's with different goals, including exploring (micro)kernel architectures [15], performance enhancements [27, 5], and better integration between protocol and application (Exokernel [7]). These works provided some important system design ideas, but have seen no real world usage because of lack of an incremental deployment path to replace legacy stacks and applications, and no support in commodity operating systems.

Instead, in this work we intentionally target commodity operating systems (Linux and FreeBSD) running on production systems, and provide support for legacy stacks alongside user-level ones. This architecture requires that (1) the namespace management system (address, protocol and port) be shared between the legacy stack and user-space stack(s), and that (2) packet I/O between the NIC and each of these stacks be isolated.

These mechanisms seem to impose significant overhead. For instance, [2] reports that mediating packets for a similar purpose with a conventional packet filter degrades performance by as much as four times. Application developers are likely to adopt user-level stacks in order to leverage new features, but not if such adoption would result in poor performance. Thankfully, recent research in the area of fast packet I/O mechanisms [23, 6] on commodity hardware and operating systems make us think that the time is ripe to make deployable, efficient user-level stacks a reality.

In this paper we introduce MultiStack, an architecture that provides operating system support for fast, user-space stacks. MultiStack builds on, and extends, two components: the netmap [23] framework and the VALE software switch [24] [†]. It combines several features that we consider essential to make it an effective and practical system: i) support for multiple independent stacks, so that applications can be upgraded independently; ii) in-kernel mux/demux, allowing centralized control and filtering of traffic; iii) extremely high performance; and iv) a fall-back path to the in-kernel network stack to support the coexistence of legacy applications and novel ones. MultiStack provides the traffic isolation needed to support user-level stacks without having to incur the costs associated with virtualization or more lightweight container technologies [17].

Our tests show that MultiStack can demux/switch packets at line rate on 10 Gbit/s at all packet sizes, and an HTTP+TCP implementation running on top of MultiStack outperforms both `nginx` and the same HTTP server running on the host TCP stack by 18–90% depending on request size.

2. DESIGN SPACE

Making multiple applications share the single I/O channel provided by the system requires managing access to the namespace (addresses, protocols and ports), preserving isolation among applications, and implementing demultiplexing of traffic with adequate speed. Here we explore these issues and the design space with respect to high performance user-level stacks.

[†]Another option would have been to use Intel DPDK's vSwitch [11], but this software cannot be used in conjunction with the host stack.

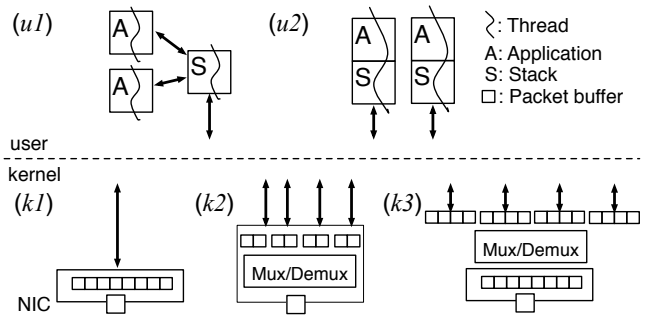


Figure 2: Options for user-space stacks (u1, u2), and network I/O (k1, k2, k3). Useful combinations are u1+k1, u2+k2 or u2+k3, the latter being our choice for MultiStack.

2.1 Shared versus Dedicated Stacks

The literature has presented two models of how a user-space stack would talk to applications and the kernel: *shared* and *dedicated* stacks. In the first model (*u1* in figure 2) the protocol stack is a process *shared* by a set of applications, as in [14]. While it works well for in-kernel stacks, applying this model to userspace introduces redundant context switches, message copies and process' hand-offs that affect both throughput and latency and are unacceptable at high data rates.

A shared stack is also problematic when it comes to resource allocation, as it makes it hard to charge network processing costs to the process that makes use of the stack [4]. While today's major operating systems do not charge all the network processing costs to the proper application (e.g., interrupt costs are not charged, for one), proper accounting will become crucial in the near future to ensure fairness between applications using higher available networking speeds and incurring the larger packets processing costs of newer TCP extensions (e.g., MPTCP to cope with middleboxes and heavy packet reordering [21] and TcpCrypt [2] to perform cryptographic operations).

The second model bundles each application with a *dedicated* stack within the same process' context (*u2* in figure 2). This removes many of the sources of overhead of a *shared* stack, and possibly enables some performance optimizations, such as improved data locality and integrated processing between the protocol and the application. Additional advantages of this model are that individual protocol stacks can be different and tailored to the specific requirements of the application, and that they can be easily updated together with the application themselves.

Dedicated stacks are therefore preferable, and we adopt them in MultiStack. However, using multiple dedicated stacks requires isolation of packet buffers, because each stack independently accesses the device driver and NIC; we describe how we address this problem next.

2.2 Sharing the Network Interface

With dedicated stacks, sharing a NIC (and its queues and

packet buffers) between multiple user-level stacks can be done with hardware support (*k2* in Figure 2), or through a software multiplex/demultiplex module in the kernel (*k3* in Figure 2). The former is adopted by Solarflare [25] and U-Net [5], and the latter is adopted by Exokernel [7].

Some modern NICs can indeed expose multiple transmit/receive queues (and corresponding packet buffers) to the device driver, and this feature can be used to assign queues to the individual user-level stacks. Standard memory protection mechanisms, or copies in the user-kernel transitions, avoid that processes access each other's traffic.

Demultiplexing is however challenging, as the requirements (by MAC address, IP address, 3- or 5- tuple) vary, and the hardware required to support this operation (CAM) is expensive. For this reason NICs often provide only a very small number of *exact* filters, which can match specific header fields and dispatch traffic accordingly [‡]. Overall, we do not consider hardware-supported mux/demux to be viable: it depends on specific hardware features that are not always present, and it has severe scalability issues due to the small number of queues and exact filters available.

The model in *k3* is better suited for user-level stacks. However, since this approach imposes software-based demux filtering as well as packet copies between NIC and application packet buffers, one obvious question arises: is it possible to achieve good performance? Fortunately, the answer is positive, and so MultiStack adopts this approach. As we will show in section 4, MultiStack achieves 60% of 10Gb/s line rate using a single CPU core, and 100% using two CPU cores even for minimum-sized packets.

2.3 Namespace Sharing

Namespace Sharing refers to the ability for stacks to share the IP addresses assigned to the NICs while avoiding clashes and information leak. Namespace sharing is achieved in today's stacks by having applications use the `bind` call which specifies the protocol, IP address(es) and port number. These three-tuples are used to dispatch incoming traffic (e.g., TCP SYN packets) to the right process.

To support multiple stacks, we use a namespace sharing module in the kernel for the same purpose (named Mux/Demux in Figure 2 (*k3*)). User-level stacks register the desired 3-tuples by executing the equivalent of the `bind` syscall on the namespace sharing module. The 3-tuples are used to demultiplex incoming data, and also serve to validate outgoing traffic for each application/stack instance.

2.4 Support for Legacy Stacks/Applications

Support for incremental and partial deployment is a fundamental prerequisite to achieve speedier protocol innovation. This seems to be one of the reasons why previous proposals of this type, although extremely interesting in terms of architecture or performance, have not been deployed.

In MultiStack, dedicated user-space stacks run in parallel with the regular host stack, ensuring support for unmodified legacy applications. To allow coexistence, we need to ensure that the kernel mechanism that handles namespace isolation is aware of which packets to deliver to the regular host stack and which to user-level ones, and to prevent port and protocol clashes between these.

Avoiding clashes requires interaction with the host stack to find out which ports are already reserved. Such information can be trivially achieved by modifying host stacks, but the changes are cumbersome. Fortunately, this information can also be obtained without modifying host stacks by either calling the socket lookup routine for receiving packets, or checking the table of file descriptors that the current process is opening, depending on the OS's implementation.

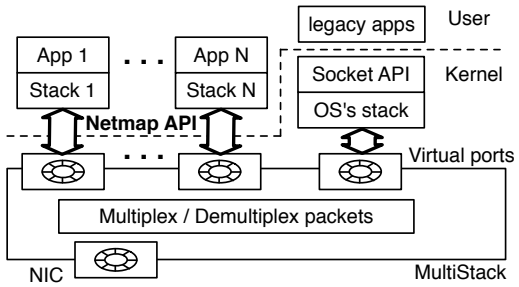


Figure 3: MultiStack software architecture.

3. IMPLEMENTATION

We have explored the design space for user-space stack support and arrived at a set of design choices: (1) support for a large number of dedicated stacks (one per network application), (2) per stack, isolated access to the NIC, (3) 3-tuple namespace isolation, and (4) support for legacy (in-kernel) stacks and applications.

As it turns out, close inspection reveals that most of these requirements point to a high-speed software switching architecture, where the switch's ports are used to provide isolated access to user-level stacks, the host stack, and the actual NICs; the actual switching logic would be based on three-tuple namespace identifiers (figure 3).

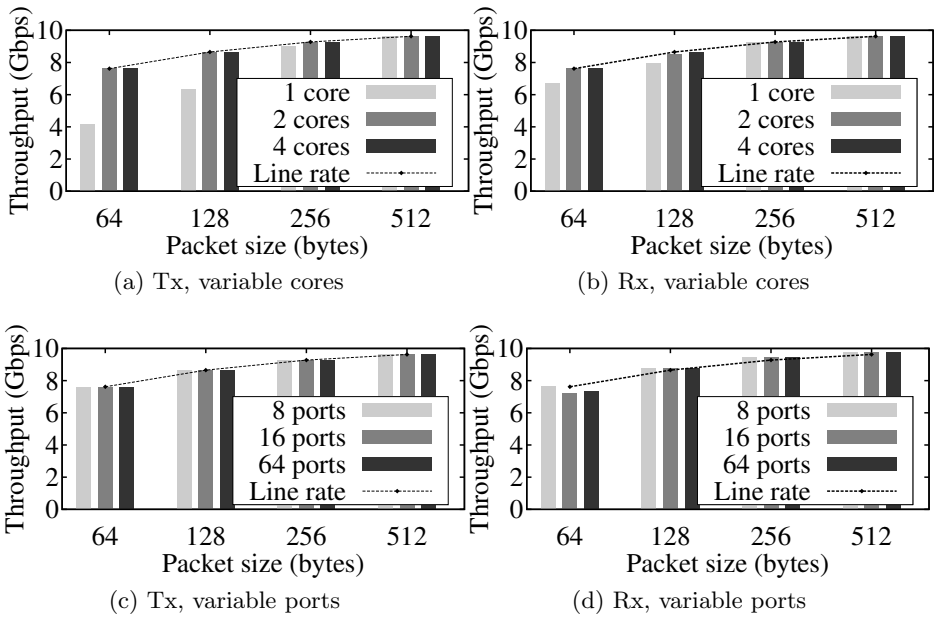
Software switches are certainly not new, and so our first task is to see if we can take an existing one as the basis for MultiStack. Perhaps the most obvious choice would be Open vSwitch [18], a multi-layer, open source software switch available upstream in the Linux kernel. Unfortunately, as a result of inefficient APIs, data structures, and packet switching logic, Open vSwitch yields poor performance. In measurements performed on an Intel Xeon 3.2 GHz box we were only able to obtain 43% of 10Gb/s line rate for 512-byte packets and 7.6% for minimum-sized packets between two NICs [§].

Consequently, we decided to base the implementation of MultiStack on VALE, a high performance software learning bridge [24], and extend it in a number of ways in order to implement our design choices. First, VALE only supports virtual ports (i.e., ports to which we would attach user-level stacks), so we add the ability to connect NICs directly to the switch. Second, we extend its functionality so that it is possible to send a subset of packets to the host stack.

Next, we replace the learning bridge logic with one that uses 3-tuples. A hash table stores three-tuple identifiers registered by the user-level stacks and is used to dispatch packets coming from the NIC. Traffic not claimed by any of the applications goes to the OS stack. Such matching

[‡]NICs also support dispatching based on a hash of selected header fields, but this mechanism is designed to help load distribution on multi-core machines and is not suitable for our purposes.

[§]Performing the same test between two user-level ports or between a NIC and a port would have resulted in worse performance since this would have incurred system call overheads.



(a) Tx, variable cores

(b) Rx, variable cores

(c) Tx, variable ports

(d) Rx, variable ports

Figure 4: MultiStack Tx/Rx performance on a 10 Gbit NIC, for variable number of cores and active switch ports. Line rate (which depends on the packet size) is achieved for all packet sizes with just 1 or at most 2 cores. Throughput is essentially unaffected even with a large number of ports sending/receiving traffic.

only applies to incoming packets; outgoing ones are just validated so that their 3-tuple (source IP, port and protocol type) matches what the stack had previously registered with MultiStack. This module also implements the registration mechanism described in Section 2.4

In addition, we implement support for multiple packet rings *per port*, each of which can be assigned to a different thread and thus a CPU core. This extension significantly contributes to performance, as we show in the next section.

We also implement a new packet forwarding algorithm that allows us to scale to larger number of switch ports, from 64 in the original VALE to 254 in MultiStack, and further if needed. The default VALE switching algorithm is optimized for multicast traffic, but this leads to a forwarding complexity that is linear in the number of connected ports (even for unicast traffic and in the presence of idle ports). To reduce this complexity, we only allow unicast or broadcast packets (multicast is mapped to broadcast). This allows us to implement packet forwarding by only scanning two lists (one for the current port, one for broadcast), which makes this a constant time step, irrespective of the number of ports. This greater port density is important if we hope to support many concurrent user-level stacks/applications.

Our implementation of MultiStack runs in both Linux and FreeBSD.

4. EVALUATION OF MULTISTACK

In this section we benchmark MultiStack's Tx and Rx throughput performance. The experiments are conducted between a MultiStack machine and an external machine connected via a direct cable through a 10 Gbit NIC. The MultiStack machine has an Intel Core i7 CPU (3.2 GHz, 6 cores, 12 with Hyper-Threading), 8 GB RAM, and an Intel 82599EB 10 Gbit/s NIC, and runs MultiStack/FreeBSD 10. The external machine has an Intel Core i7 CPU (3.4 Ghz, 4 cores), 4 GB RAM, and the same NIC and OS.

The MultiStack ports are connected to either a custom packet counter (Rx), or to a generator (Tx). The packet generator is multi-threaded and emulates TCP's basic behavior, building each packet from scratch, incrementing its sequence number, and calculating a checksum for it. The complementary program (Tx or Rx, respectively) runs on the remote machine, this time using the netmap API directly on the NIC. We should note that the switching code in MultiStack also performs basic IP header validation (header length, protocol type, IP checksum). While not strictly necessary, it gives us a more conservative estimate of the throughput.

The first two tests measure the Tx (figure 4(a)) and Rx (4(b)) throughput when only one MultiStack port is in use (each port supports multiple Tx/Rx queues, so we can use multiple cores in the sender and the receiver if needed). One core is sufficient to saturate the link for all but the smallest packet sizes, and two are sufficient in all cases. These numbers are slightly worse than those of the individual netmap and VALE components on the same hardware, but we should remember that here we have a lot more processing in the sender, switch and receiver.

The next set of experiments (figures 4(c) and 4(d)) measures the Tx and Rx throughput with multiple active MultiStack ports. Each sender or receiver connects to a MultiStack port using a single thread, and we let the OS handle CPU assignment (which is suboptimal, but again should give conservative numbers). In the Tx experiments we have not used any mechanism to guarantee short term fairness among the greedy senders, though 1 sec statistics indicate roughly equal rates. Consistent with figure 4(a), we see that line rate is reached for all packet sizes.

In the Rx experiments, the source spreads packets evenly to all destinations, one 3-tuple per MultiStack port. Since the total amount of packets is constant (i.e., it is limited by the 10Gb/s pipe), increasing the number of ports means that each port gets fewer packets per system call, where each system call has a certain overhead. This explains why we are slightly below line-rate for 64-byte packets and larger number of ports.

5. EVALUATION WITH A USER-SPACE STACK AND APPLICATION

The main goal of MultiStack is to allow new network stacks or features to be more easily (and quickly) tried out and deployed. However, this will only happen if such stacks can yield good performance. In the previous section we showed that MultiStack would not be the bottleneck, so the question now is whether we can implement an efficient user-level stack on top of it.

One might suggest simply running stacks extracted from the kernel in user-space [13, 19]. However, this approach would hinder the performance evaluation of MultiStack, since it results in two orders of magnitude lower performance than the rates presented in the previous section Instead, we carry out our investigation of user-space stacks by developing, from scratch, UTCP, a simple TCP implementation which is fully aware of MultiStack (e.g., packet batching and integration with an event loop of the netmap API).

5.1 UTCP

UTCP is a simple user-level TCP library that we developed and that can be linked to an application. Its API boils down to a few functions: `tcp_input()` to process a packet received, `tcp_output()` to add data to an outgoing connection, and `tcp_close()` to close a connection. The `poll()` system call is used to send and receive packets to the MultiStack port, possibly in batches.

Figure 5 shows basic packet reception code that an application running on top of UTCP can use. From the main event loop, each input packet goes to `tcp_worker()`, which handles basic functions (SYN/ACK/FIN processing, among others). Depending on the return code we then may have a chance to process new incoming data.

In this paper we use these relatively low-level APIs in order to see the ideal performance of user-space stacks, that is, without incurring overheads that would arise from things like socket API emulation. In Section 6 we discuss the inclusion of more practical APIs in MultiStack in order to encourage adoption of user-space stacks.

```
int tcp_event_loop(tcp_worker_t *worker) {
  struct pkt_bufs *bufs = &worker->pkt_bufs;
  while (1) {
    poll(worker->fd); /* get new packets */
    for (; bufs->unprocessed; next_pkt(buf))
      tcp_worker(pkt_buf(bufs), worker);
  }
}
/* Process a single packet */
void tcp_worker(char *pkt, tcp_worker_t *w){
  /* Parse pkt, init TCP for new conn. */
  int status = tcp_input(pkt, w->tcbs);
  if (status == NEW_CONNECTION)
    ; /* no-op, TCB already set in tcp_input() */
  else if (status == NEW_DATA)
    process_data(w);
  else if (status == NEW_ACK) {
    if (w->data.off == w->data_eof)
      tcp_close(w->tcbs->cur);
  }
}
```

Figure 5: Pseudo-code for packet reception using UTCP.

5.2 Evaluation: Simple HTTP Server

We tested the performance of UTCP by building a simple HTTP server on top of it. The application-specific logic, implemented in `process_data()`, is marginal: a single strcmp() to validate HTTP GETs, and a bit more code to reply with the HTTP OK header and payload usi ng `tcp_output()`, which then does the necessary segmentation. An HTML file served as the HTTP OK is mmap()ed to UTCP's address space at HTTP server initialization time. . The application data is thus copied twice: once from application/stack into the back-end switch's virtual port, and once from it to a NIC packet buffer.

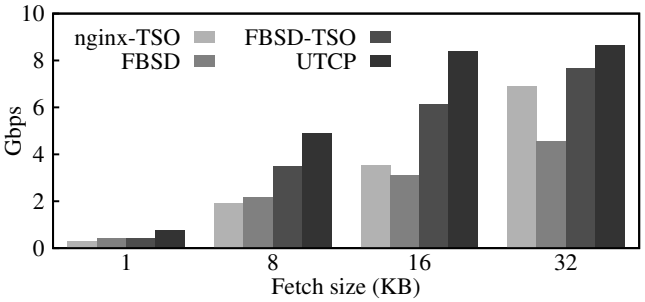


Figure 6: Performance of a simple HTTP server running on top of UTCP compared to other solutions using the host stack (single CPU core).

We are interested in the connection setup and data transmission performance, so we issue a single HTTP transaction per connection while varying the payload size between 1 and 32 KB [¶]. The smaller fetch size is useful to measure connection setup performance, and the larger one is useful to evaluate data transmission performance.

In figure 6 we compare the performance of four different configurations: our server on top of UTCP; the same server on top of the socket library (the functions in our simple HTTP server can be trivially implemented on top of the socket library) running on the OS stack with or without TCP Segmentation Offloading (FBSD-TSO and FBSD respectively); and `nginx`. All configurations use one core, and ten packets for the initial window size. Our client is wrk [8], a fast, multi-threaded HTTP benchmark tool. The hardware setup is the same as in section 4.

The results (figure 6) confirm that the efficient datapath provided by MultiStack supports very high throughput, up to 8 Gbit/s (and approx 100 K requests/s for short requests) on a single CPU core, without using any NIC acceleration nor exploiting application knowledge. At least for these transfer sizes (relevant to HTTP servers), UTCP outperforms the host TCP stack, even when the latter is using TSO.

6. DISCUSSION

We have presented results that show MultiStack's high performance when running user-level stacks and applications on top of them. Such performance derives from two main factors. The first is a result of the streamlined data path from NIC to user-level stack and the fact that MultiStack's isolation and mux/demux mechanisms incur little overhead. Second, our UTCP stack foregoes the expensive connection setup costs associated with the socket API (alloc/dealloc of file instances, inode entries and unassigned file descriptors [9]). Indeed, comparing UTCP to FBSD-TSO we see a 90% speedup with a 1KB fetch size, where a relatively

[¶]A recent study claims that 73% of TCP connections deliver less than 4,380 bytes of data [1].

long time is spent on connection setup, and a 58% speedup with a 8KB fetch size, where a smaller fraction of the time is spent on that operation.

The last point raises the issue of what is a proper API for applications. Emulating the socket API clearly makes user-space stacks amenable to existing applications; however, exactly reproducing the semantics of the socket API (e.g., with respect to `fork()`, see [25]) is non-trivial and would have a negative impact on performance. For this reason we do not try to emulate the socket API completely, but believe that providing reasonably similar, generic APIs would allow for relatively easy adoption of user-level stacks while retaining high performance. There are a number of reasons in support of this decision:

- MultiStack already supports legacy applications through the host stack.
- Applications already leverage network API wrapper libraries such as libevent and libuv, which provide higher-level functionality and hide OS-specific features (e.g., `epoll()` in Linux and `kqueue()` in MacOS and FreeBSD).
- The quest for performance has historically pushed for new APIs (e.g., `sendfile(), epoll()` and recent proposals such as MegaPipe [9]), and developers are generally open to using better, more performant APIs even if it means having to adapt applications.

We are currently in the process of evaluating which such wrapper libraries to include in MultiStack in order to further increase the likelihood of user-level stack adoption.

7. CONCLUSION

In this paper we have argued for the adoption of user-level network stacks to accelerate the evolution of protocols. While MultiStack requires the inclusion of a kernel module, this change to the kernel is only needed *once*, after which deployment of new features can be easily and quickly be carried out in user-space.

Our experiments show that our MultiStack prototype can switch minimum-sized packets at 10Gb/s while supporting a substantial number of user-space stacks. To showcase the benefits of MultiStack, we have also implemented UTCP—a user-level TCP stack—and a simple web server. The HTTP server/UTCP combination runs 18-90% faster than similar applications using the host stack, despite the fact that MultiStack does not (yet) support hardware offload.

Future work will address the design of a more complete user TCP stack implementation. At the moment UTCP is just a proof of concept, and most existing user space TCP prototypes are equally incomplete, old and not necessarily designed to exploit batching and achieve high speed. We also plan to work on the design of APIs for user-space protocol stacks, possibly adapting and adopting existing networking libraries such as libuv and MegaPipe as well as socket-like APIs. This will make it easier for new and existing network applications to adopt user-space stacks.

Finally, we also plan to support the offloading features available in modern NICs. While we showed that user-space stacks without TSO can achieve higher performance than an in-kernel stack with TSO, it makes sense to leverage offloading capabilities where available. These would be complementary to the kernel, network stack and API optimizations presented in this paper.

8. REFERENCES

[1] M. Allman. Comments on buffer bloat. *ACM CCR*, 43:31–37, 2013.

[2] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. The case for ubiquitous transport-level encryption. In *Proc. USENIX Security Symposium*, Aug 2010.

[3] K. Cho, K. Mitsuya, and A. Kato. Traffic data repository at the WIDE project. *USENIX ATC*, 2000.

[4] P. Druschel and G. Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *Proc. USENIX OSDI*, Oct. 1996.

[5] T. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. ACM SOSP*, pages 40–53, 1995.

[6] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Proc. ACM IMC*, pages 218–224, 2010.

[7] G. Ganger, D. Engler, M. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM ToCS*, 20(1):49–83, 2002.

[8] GitHub. Modern HTTP benchmarking tool. `https://github.com/wg/wrk`, July 2013.

[9] S. Han, S. Marshall, B. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proc. USENIX OSDI*, Oct. 2012.

[10] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it Still Possible to Extend TCP? In *Proc. ACM IMC*, pages 181–192, 2011.

[11] Intel Open Source Technology Center. Intel DPDK vSwitch. `https://01.org/packet-processing/intel-ovdk`, 2014.

[12] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC 1323*, May. 1992.

[13] A. Kantee. Environmental Independence: BSD Kernel TCP/IP in Userspace. *AsiaBSDCon*, 2009.

[14] C. Maeda and B. Bershad. Networking performance for microkernels. In *Proc. IEEE WOS workshop*, pages 154–159, 1992.

[15] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *Proc. ACM SOSP*, pages 244–255, 1993.

[16] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, Oct. 1996.

[17] Microsoft Research. Drawbridge. `http://research.microsoft.com/en-us/projects/drawbridge/`, 2014.

[18] OPEN VSWITCH. Open vSwitch. `http://openvswitch.org`, 2013.

[19] B. Penoff, A. Wagner, M. Tuxen, and I. Rungeler. Portable and Performant Userspace SCTP Stack. In *Proc. IEEE ICCCN*, pages 1–9, 2012.

[20] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. Tcp fast open. In *Proc. ACM CoNEXT*, December 2011.

[21] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proc. USENIX NSDI*, 2012.

[22] I. Rhee and L. Xu. Cubic: A new tcp-friendly high-speed tcp variant. *Proc. PFLDNeT*, 2005.

[23] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proc. USENIX ATC*, 2012.

[24] L. Rizzo and G. Lettieri. Vale: a switched ethernet for virtual machines. In *Proc. ACM CoNEXT*, December 2012.

[25] SolarFlare. OpenOnLoad. `http://www.openonload.org`, 2013.

[26] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, pages 1–12, 2006.

[27] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. *IEEE/ACM ToN*, 1(5):554–565, 1993.

[28] Wikipedia. QUIC. `http://en.wikipedia.org/wiki/QUIC`, 2014.

[29] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proc. USENIX NSDI*, 2011.