Slipstream: Automatic Interprocess Communication Optimization

Will Dietz, Joshua Cranmer, Nathan Dautenhahn, and Vikram Adve,
University of Illinois at Urbana-Champaign

https://www.usenix.org/conference/atc15/technical-session/presentation/dietz

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Slipstream: Automatic Interprocess Communication Optimization

Will Dietz, Joshua Cranmer, Nathan Dautenhahn, and Vikram Adve
University of Illinois at Urbana-Champaign
{wdietz2,cranmer2,dautenh1,vadve}@illinois.edu

Abstract

We present **Slipstream**, a userspace solution for transparently selecting efficient local transports in distributed applications written to use TCP/IP, when such applications communicate between local processes. Slipstream is easy to deploy because it is language-agnostic, automatic, and transparent. Our design in particular (1) requires no changes to the kernel or applications, (2) correctly identifies (and optimizes) pairs of communicating local endpoints, without knowledge of routing performed locally or by the network, and (3) imposes little or no overhead when optimization is not possible, including communication with parties not using our technology. Slipstream is sufficiently general that it can not only optimize traffic between local applications, but can also be used between Docker containers residing on the same host. Our results show that Slipstream significantly improves throughput and latency, 16-100% faster throughput for server applications (and 100-200% with Docker), while imposing an overhead of around 1-3% when not in use. Overall, Slipstream enables programmers to write simpler code using TCP/IP "everywhere" and yet obtain the significant benefits of faster local transports whenever available.

1 Introduction

TCP has become one of the most commonly used communication protocols because of its ubiquity on standard platforms (e.g., Windows, Android, Linux) and its *location transparency*: instead of creating one communication channel for host-local and another for remote communications, which would reduce portability and increase complexity of the application, developers use TCP because it works for all cases. Unfortunately, by using TCP, developers eschew faster host-local transport mechanisms (e.g., Unix domain sockets, pipes, or shared memory) resulting in missed performance opportunities: a claim supported by a comprehensive study providing clear evidence for the *potential* improvements to be had by replacing the TCP transport with other local IPC mechanisms [27].

Using TCP for its *location transparency* to reduce programming burden and enhance portability is common in several application domains. Web sites are commonly deployed on a single system, yet the Web server front-end communicates with database engines and application servers over TCP, hurting both latency and throughput of Web requests (e.g., LAMP). In some instances, application logic depends on TCP parameters [16], e.g., Memcached uses the IP address and TCP port to implement consistent hashing [10], thereby requiring TCP addressing to function correctly, but *not* necessarily TCP data transport.

Perhaps the most compelling future need for optimizing local TCP communication is the increasing popularity of lightweight virtualized environments like Docker. Docker strongly encourages separating communicating services into distinct Linux containers, using TCP to communicate with each other because portability is a key goal [21]. For example, Yelp developers created a service discovery architecture in which client applications communicate with remote endpoints through a host-local proxy (`haproxy` [1]) via TCP [8]. One of the primary deployment scenarios for this architecture is to run a client application and its local proxy in separate containers, which puts the TCP latency on the critical path for every client request and response. Optimizing local TCP communication over Docker can therefore provide important performance gains (as our experiments with other containerized Docker services show).

In fact, the problem is important enough that there are several approaches that optimize TCP communication between communicating processes within single operating system environments. This includes commercial operating systems like Windows [5], AIX [23] and Solaris [19] and several userspace libraries [2, 25, 29]. However, these approaches either require changes to the operating system [5, 19, 23] or application code [2, 25, 29]—thus eliminating one of the key benefits of using TCP in the first place—or they are only applicable to specific language runtimes. In legacy deployments, it might not be possible to modify the OS or application, and furthermore, in cases where modifications are possible, they may not be feasible: any modifications may cost too much to make to existing application logic.

We present **Slipstream**, a userspace solution that identifies and optimizes the use of TCP between two host-local communicating endpoints without requiring changes to the operating system or applications (except for the use of a shim library above `libc`). Slipstream transparently reduces latency and increases throughput without requiring modifications to either the kernel or the application by interposing on TCP related events to detect and optimize the communication channel.

We built a Linux prototype of Slipstream that detects TCP-based host-local communications by inserting an optional shim library above `libc` to intercept TCP communication operations.[1] Our solution is portable across UNIX-like systems. Slipstream uses this vantage point to collect information on connections in order to apply a general detection algorithm that relies only on observable characteristics of TCP, without knowledge of the underlying network topology, to detect host-local communication endpoint pairs. Once a host-local TCP communication stream is detected, Slipstream transparently replaces TCP with a faster host-local transport while emulating all TCP operations to maintain application transparency. The primary complexity in the design and implementation of Slipstream arises in replicating kernel-level TCP state at the user-level and preserving the interface semantics of the TCP sockets API on top of the host-local transport.

Our results indicate significant performance benefits for applications using Slipstream: throughput improves up to 16-100% on server applications and 100-200% with Docker, and microbenchmarks show that latency is cut in half. Our results also show that when Slipstream tries but fails to optimize a connection (e.g., because one of the two endpoints is not using Slipstream), the throughput is impacted by only 1-3% on average. Moreover, Slipstream is an opt-in system that imposes *zero* overhead for applications that do not explicitly request the optimizations.

Our work makes the following contributions:

- We describe a novel backwards-compatible, transparent algorithm for classifying communication between two endpoints as host-local or remote.
- We describe a fully automatic optimization to replace TCP with Unix domain sockets as the transport layer, while preserving the interfaces, reliability guarantees, and failure semantics of the original transport.
- We show by use of microbenchmarks and server applications that Slipstream achieves significant improvements in throughput, without requiring manual tuning, custom APIs, or special configuration.

There are certain system configurations that will cause our system to mismatch connections; violations of our correctness conditions, while unlikely in practice, must be avoided during system setup.

Overall, our experience suggests that the improvements achieved automatically by Slipstream are comparable in terms of performance (as we show in the Netperf results) to those that can be achieved by modifying applications to explicitly use Unix domain sockets for host-local connections.

2 Slipstream Overview

Slipstream transparently identifies and dynamically transforms TCP streams between local endpoints into streams that employ more efficient IPC mechanisms, such as Unix domain sockets (UDS). This optimization improves communication performance for many existing applications, and alleviates the burden on programmers to manually detect and select the fastest transport mechanism. To accomplish this task, Slipstream interposes on TCP interactions between the application and the operating system to track TCP endpoints, detect local TCP streams, switch the underlying transport mechanism *on-the-fly*, and emulate TCP functionality on top of the local transport mechanism.

Performing all of these in userspace means that Slipstream must replicate critical stream state at the userspace level and it must adequately emulate TCP on a non-TCP-based transport mechanism. In this section we describe the key design goals we aim to meet, discuss the challenges presented by TCP, and then provide a high level description of Slipstream.

2.1 Design Goals

We specify three key design goals for the optimization, which are desirable for real-world use and present new design challenges. None of the previous systems meet all three requirements. First, we aim to preserve application transparency, i.e., requiring *no changes* to application code in order to perform this replacement. However, application end-users can choose whether or not to enable the optimization. Second, we aim to avoid any operating system changes, not even through dynamically loadable kernel modules, i.e., the optimization should be implemented entirely via userspace code (in fact, we do not even require root privileges). Although aspects of the optimization would be simpler to implement within the OS, those extra challenges are solvable in userspace as well (as we show), and a solution that does not require kernel changes is far easier to deploy. Third, unoptimized communication (i.e., between an optimized component and an unoptimized one) must continue to work correctly and, again, with no application changes.

Of course, the system must also meet essential correctness requirements: the reliable stream delivery guarantees of TCP must be preserved, and the semantics of socket operations must be implemented correctly.

[1] The source code for Slipstream and the scripts used in the evaluation are available at: `http://wdtz.org/slipstream`.

2.2 TCP Optimization Challenges

Within an OS, a *TCP endpoint* is a kernel-level object represented as a 2-tuple, $\langle$local IP address, TCP port number$\rangle$. A TCP stream is a pair of *TCP endpoints*, and is also referred to as a *socket pair*. The kernel provides userspace access to TCP via the standard socket interface, which applications use for creating and managing connections and for sending and receiving data. The socket interface represents instances of streams in the form of a socket descriptor, a special case of a file descriptor. These file descriptors are the only way in which userspace code can access TCP endpoints. Since Slipstream is a userspace mechanism but must emulate details of the TCP protocol, it operates by replicating the notion of a TCP endpoint in userspace.

TCP, the POSIX socket interface, and their implementation in modern operating systems present some key design challenges:

- Although a socket pair uniquely identifies TCP connections in a network, a single host can be part of multiple networks with overlapping network name spaces. For example, via use of virtual environments, it is possible to have the same IP address assigned to multiple network interfaces within the same system. This allows duplicate combinations of IP address / TCP port pair to be used for distinct TCP streams within the same host. Each such combination would be a distinct TCP endpoint in the kernel.
- It is common for the kernel to map a single TCP endpoint into multiple process address spaces, thereby creating several userspace file descriptors for a single kernel-level TCP endpoint. This feature is widely used in applications, such as Apache. Consequently, Slipstream must also track all application interactions with the kernel that might create or delete multiple instances of such endpoints.
- The TCP protocol does not support any reliable mechanism for transferring extra data "out-of-band" between the endpoints[2]. On the other hand, injecting any such data into the stream "in-band" would break an application that isn't using Slipstream and so cannot filter out the extra data. This significantly complicates the task of detecting when two endpoints are on the same machine and capable of using Slipstream.
- POSIX file descriptors support a large number of non-trivial functional features through numerous system calls, which must be correctly emulated to preserve application functionality transparently. Some are specific to sockets (e.g., `bind`, `connect`, `listen`, `accept`, `setsockopt`) while others are generic to file descriptors (e.g., `poll, dup, dup2, fcntl, fork, exec`, and the various send/receive operations). Slipstream supports all of these system calls and other less common ones that interact with TCP.

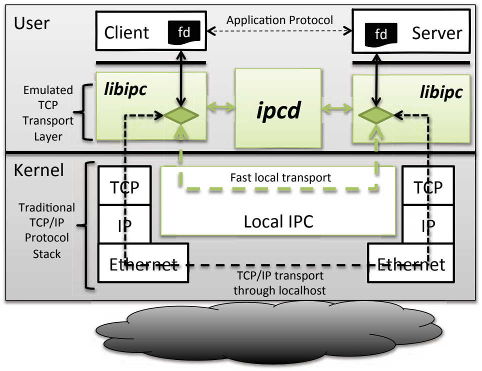


Figure 1: Network layers and local IPC within Slipstream.

2.3 Slipstream Overview

Figure 1 shows a high-level diagram of a typical OS network stack, enhanced with Slipstream. Slipstream inserts a shim library we call `libipc` that interposes on all TCP interactions between the application and the kernel. `libipc` is responsible for tracking TCP endpoints at all TCP-relevant system calls and for reporting all TCP stream identifying information to a system-wide process, `ipcd`. `ipcd` collects and records all stream information and analyzes all existing streams using a stream matching algorithm.

Once Slipstream detects that both endpoints of a stream are local, `libipc` modifies the underlying communication channel to use a local IPC transport (in the case of our implementation, UDS). The use of emulation also indicates one of the major contributions of our efforts: emulating a sufficient subset of the TCP protocol in userspace to correctly support real applications, as demonstrated in our evaluation.

Overall, this sequence of steps ensures that (a) Slipstream can replace TCP with a local IPC transport without requiring any changes to application code; (b) Slipstream does *not* break remote streams or local streams that cannot be identified by Slipstream; and (c) the protocols used by Slipstream *never* introduce new errors in communication for identified local streams. In this sense, the optimizations are *transparent* to application code, are *backwards-compatible* with non-participating endpoints, and do not require kernel modification.

3 Design and Implementation

The functionality of Slipstream has three major aspects: (1) identifying TCP communication streams in which both endpoints are local; (2) replacing TCP with an alternative local transport; and (3) emulating most of the functionality

[2] There is a TCP urgent data feature, but its use to communicate lengthy amounts of data is unreliable.

of the TCP sockets interface. The first two subsections below describe preliminary design aspects for interposing on and tracking TCP events. Subsections 3.3– 3.5 then discuss the three major aspects of the design.

3.1 Interposing on TCP Events

In order to identify local streams and emulate TCP on optimized transport, we monitor applications and track the creation and use of TCP endpoints. We do this using our per-process library, `libipc`, which intercepts all TCP-related calls exposed by the system. In our implementation, we insert `libipc` into a client application by using the `LD_PRELOAD` environment variable or by specifying the full path to `libipc` in `/etc/ld.so.preload`. We prefer dynamic interposition in favor of replacing `libc` so as to avoid requiring modifications to the system libraries and, importantly, to enable applications to choose whether or not to use our technology.

3.2 TCP Endpoints in Userspace

In order to track TCP streams, Slipstream assigns a unique *endpoint identifier* (*EPid*) to each TCP endpoint created and used by the application. Each *EPid* represents an in-kernel TCP endpoint. To manage the optimization state, `libipc` assigns a state to each *EPid* to track its optimization status:

Pre-opt	Optimization not yet attempted.
No-opt	Optimization attempted and failed.
Opt	Optimization successful.

As explained in Section 2.2, in order to fully track streams, Slipstream must replicate endpoint state at the user level (in `libipc`) by tracking TCP state at critical system calls, such as `fork`, as well as traditional TCP modifying operations. For each *EPid*, `libipc` maintains a reference count representing how many processes have at least one file descriptor open to this endpoint. `libipc` updates this reference count on events that affect it, such as `fork`, `exec`, and `exit`. Moreover, instead of communicating with `ipcd` on every use of a socket, as many details as possible about file descriptors and *EPid*s are retained by `libipc`.

In our implementation, `libipc` state information is maintained in two tables, one tracking file descriptors and the other tracking endpoints. The file descriptor table tracks much of what the kernel also tracks at a per-file descriptor granularity, such as the `CLOSE_ON_EXEC` flag or `epoll` state. In addition, this table also tracks the mapping of file descriptors to endpoint identifiers. The *EPid* table tracks the optimization state for each *EPid*, explained above. It also tracks information that is relevant for the optimization procedure, such as the handle for local transport if optimized and running hashes of sent and received data.

3.3 Identifying Host-Local Flows

The first major step of Slipstream is to identify when two endpoints of a TCP stream are located on the same host. As noted in Section 2.2, the combination of IP address and TCP port is insufficient to do so because it is possible to have access to two network domains on a single host, even though this may be rare. Instead, to identify local TCP streams, Slipstream augments the usual IP address and port pairs with extra information passively obtained by watching the initial TCP conversation. This information consists of hashes of the first N bytes of the stream and precise timing of the connection creation calls. Together, these components are sufficient to pair endpoints in the vast majority of situations. When they are not sufficient, Slipstream detects such situations and does not attempt to optimize the socket.

More specifically, the steps Slipstream takes are as follows. When a new TCP socket is connected, `libipc` immediately records the time of the connection attempt and forwards it to `ipcd` along with the IP and port information. `ipcd` uses this information (all but the hashes) to identify endpoints that are likely candidates for pairing. By receiving this information *immediately*, without waiting for the hashes, `ipcd` can eagerly detect if multiple pairings are possible due to overlapping address/port pairs and timing information. After N bytes have been sent in one direction on the stream, `libipc` contacts `ipcd` to attempt to find a matching endpoint. Since the N-byte transfer almost certainly takes significantly longer than reporting the connection information to `ipcd` for reasonable[3] values of N, this ensures that if a mis-pairing is possible *it is detected before the optimization happens*. In this case, the stream is conservatively switched to the No-opt state, and optimization is aborted.

If a single matching endpoint is found, `ipcd` initiates the optimization procedure, explained in Section 3.4. If a matching endpoint is not found, `ipcd` records the current endpoint in a list and waits for a match, while `libipc` tries again several times after which it declares the procedure has failed. In this case, `libipc` changes the state of the *EPid* to the ‘No-opt’ state. The last request by `libipc` for a matching endpoint is sent with a flag telling `ipcd` to remove the list entry if it is not matched. This removal serves two purposes: first, it helps eliminate matching errors by preventing stale endpoint data from being matched; second, the atomic “request-or-removal” avoids the issue of having only one endpoint aware of a pairing: `ipcd` only pairs endpoints if they have both indicated they will check again for a pair.

[3] In our prototype, we use $N = 2^{16}$.

3.4 Transparent Transport Switching

Once ipcd has determined that two local endpoints are communicating with each other over TCP, ipcd generates a pair of connected sockets using a faster transport (in our implementation, Unix domain sockets) and passes them to the libipc instances controlling the communicating endpoints. Generating the new sockets in ipcd and not libipc allows the procedure to work even when the two libipc instances cannot directly communicate, such as between separate Docker containers. Upon receiving its side of the faster socket, libipc copies appropriate flags from the old socket to the new socket and then changes the state of the *EPid* to 'Opt'.

libipc then migrates communication to the new channel for improved throughput and latency. The primary challenge in doing this correctly is ensuring that both endpoints will switch to the new channel at the same position in the data streams.

To make this switch, libipc ensures that a send or recv request ends at exactly *N* bytes. For a non-blocking send or recv operation on the TCP socket, libipc merely truncates this request, returns a short write and lets the application issue the next request as normal. For blocking operations, in which a short write could be misinterpreted by the application, libipc instead splits the request into two pieces and processes them internally as two requests but only returns control to the application after both requests have been processed. After splitting the request, libipc attempts to optimize the endpoint as detailed above, and subsequently transfers the remaining bits of the request using the selected transport, except that the request is handled in a non-blocking manner to better emulate what would have occurred if the entire request had been processed without libipc intervention.

3.5 Emulating TCP in User-Space

The bulk of the socket API has a straightforward implementation in libipc: whenever a file descriptor that has an underlying endpoint identifier is mentioned, the real API is called with either the optimized transport's file descriptor (if in the 'Opt' state) or the original TCP socket's file descriptor (otherwise). Some system calls, however, require a more complex implementation.

A global connection table is maintained by ipcd to coordinate the matching process. Entries are created in this global table whenever functions like socket and accept request to create a TCP socket and are initialized with properties about the socket such as the local and remote IP address. Removal from the global table only happens when a libipc instance determines that all file descriptors within that process that refer to a single endpoint have been closed. Since it is is possible to share endpoints across multiple processes via use of fork, ipcd maintains a count of the number of processes using a single endpoint identifier.

The basic read and write functions (send, recv, recvmsg, etc.) require more work when the endpoint identifier is in the 'Pre-opt' state, where the tracking of the first *N* bytes and the seamless transition steps described in Section 3.4 need to be executed in addition to the normal I/O.

Emulating fork requires more care due to the introduction of multiple processes that could potentially race on communication to ipcd. This race is resolved by having libipc inform ipcd that all of its endpoint identifiers are about to be duplicated before making the call to fork; should the call to fork fail, libipc tells ipcd to close the duplicated file descriptors. If libipc were to notify ipcd of the endpoint identifier duplication after the call to fork, then a child process that immediately calls close on its copy of the file descriptor would cause ipcd to prematurely clean up the connection.

While fork is emulated by libipc, the implementation is only sufficient to support sharing file descriptors across multiple processes if only one process at a time communicates on it. This level of support is sufficient to support most forking server applications, in which the parent creates a socket, forks, and then closes the socket while the child process does all of the communication on said socket.

The exec family of functions implicitly refer to file descriptors by the need to clean up those that have the CLOSE_ON_EXEC flag, but they also pose a challenge to support since the internal memory, including both the code and data segments of libipc, is completely wiped. libipc retains its memory across the exec call by copying it to a shared memory object tied to a file descriptor that is retained across the exec call. If the new process uses libipc, the initialization process first reads this table and initializes its current state. If the new process does not use libipc, the tie to the file descriptor at least ensures that the eventual death of the process will clean up the system resources allocated by libipc.

4 Discussion

The current design of Slipstream assumes no ability to communicate protocol data using packets within the original stream, i.e., to add reliable "out-of-band" (OOB) signaling between two libipc instances connected in a stream. This has implications for performance, correctness, and security.

4.1 Performance

The implication for performance is that Slipstream may fail to optimize some instances of host-local TCP communication, i.e., we may have *false negatives*. For example, Slipstream may conservatively decide not to

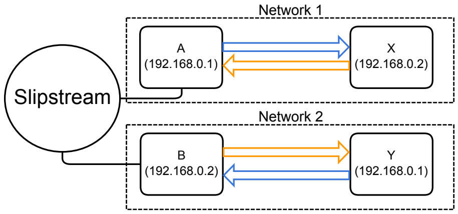


Figure 2: Parallel Network Configuration Example

optimize a stream if it exchanges fewer than N bytes, or if the match is not identified within a short time interval.

More generally, we focus on optimizing using observations external to the TCP stream itself that are readily available on any system. Moreover, we design the protocol conservatively to avoid *false positives*, which represent correctness violations. To avoid such errors, Slipstream must be able to disambiguate TCP streams in fairly complicated scenarios such as multiple identical virtual networks with endpoints having identical conversations.

4.2 Correctness

It is virtually impossible *through an accidental misconfiguration* for Slipstream to incorrectly match and optimize endpoint communication. A “mispairing” (or “false positive” or “incorrect match”) can only occur if *all* of the following are true:

1. There are multiple TCP streams described by the same 4-tuple ⟨SrcIP, SrcPort, DstIP, DstPort⟩, and there must be at least one common host system running more than one stream.
2. These streams were established with overlapping timings.
3. The first N bytes of these streams are identical.
4. Slipstream is deployed to some, but not all, of the endpoints described in these streams.

An example of a scenario that would be needed to cause false positives is shown in Figure 2. In this example, A and B are using Slipstream but X and Y are not. The ports used by A and Y (not shown in figure) must be the same, and so must be the ports used by B and X. This scenario would then satisfy condition 1 as the same 4-tuple of the IP addresses and ports identifies both concurrent TCP streams, A−X and B−Y. In addition, these endpoints must establish connections at approximately the same time (within the time window used by a `libipc` instance to poll its associated `ipcd` for a match; not greater than 100ms), and both connections must communicate the same first N bytes of data. Only if all these conditions hold will Slipstream erroneously attempt to pair endpoints A and B.

The need for the first three conditions is obvious from our endpoint matching protocol. The first condition cannot occur within a single well-behaved network; for the same IP address to occur in multiple distinct streams, there must be local endpoints that reside on distinct networks making use of the same IP addresses. Even in such scenarios, the ports used must match as well, which is unlikely because it is very common for clients to use randomly generated port numbers (called *ephemeral ports*) when setting up connections with servers, by using a range of dynamic port numbers set aside for this purpose [12].

Condition 4 is required because if Slipstream is deployed to all endpoints, the possibility of mispairing will be detected before optimization is attempted, and Slipstream will conservatively avoid the pairing.

While these four conditions are possible, they are unlikely to occur accidentally. A well-configured system would not assign identical IP addresses to different interfaces. The use of ephemeral ports, which are drawn randomly from a fairly large range (e.g., 32768-61000 by default on recent Linux kernels, for IPv4) makes condition (1) even more unlikely. The two connections using those two ports must both be started within a very small window of time. Finally, the connections must send exactly the same N bytes of data, for moderately large values of N (e.g., 2^{16}). As a result, Slipstream is well-suited for most real-world applications and is only unsafe when deployed to applications known to intentionally violate one or more conditions (e.g., regularly sending the same first N bytes).

4.3 Security

The key new security risk posed by Slipstream is that an attacker (any unauthorized third party) could try to force a mispairing, i.e., that the attacker is given read or write access to a TCP stream to which she did not have access previously. The threat model we assume is that the attacker must have local access to a machine where either endpoint of some local stream lives (necessary to talk to `ipcd`), *and* does *not* have root privileges on that machine (with root, more powerful attacks are possible even without Slipstream).

In the absence of root access, it is impossible for the attacker to forge IP headers or to misconfigure a second network to obtain duplicate IP addresses as an existing network. In our current *implementation*, however, we trust that each `libipc` is being honest in describing the information about its socket connection. A `libipc` controlled by a local attacker could simply “lie” about its IP address and port number and could potentially construct the remaining information, including the N-byte hash, in order to fool `ipcd` into giving it an optimized endpoint incorrectly. A simple solution is to give `ipcd` sufficient privileges to verify the IP address sent to it. On a well-configured system that is not running a Docker-like environment, this is sufficient to prevent a non-root attacker from impersonating another endpoint.

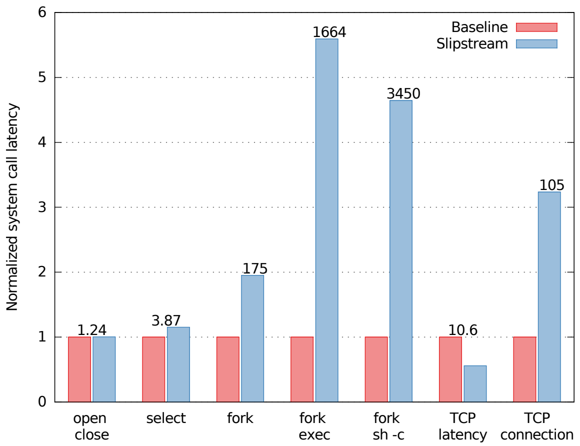


Figure 3: System call overheads. Values for Slipstream in microseconds are shown above the bars. (Lower is better.)

On a system running a Docker-like environment, it is plausible that multiple containers are assigned the same IP address in a virtual network configuration. The default Docker configuration, however, is to assign all containers unique IP addresses from a single virtual network, which prevents condition 1. Restricting Slipstream to use only in the default configuration therefore eliminates any security risk from untrusted containers. We leave it to future work to support non-default Docker configurations with duplicate IP addresses securely. For example, it might be possible to abandon our initial assumption and extend Slipstream to exchange data reliably "out-of-band" but within the TCP stream by building on existing approaches in the literature, e.g., through covert use of various TCP/IP headers [31]. This would add some complexity to `libipc`, but would be justified in large installations (e.g., a data center) in which the one-time cost of enhancing `libipc` would benefit many customers.

5 Results

To evaluate Slipstream, we use a suite of microbenchmarks and applications that measure the performance of various aspects of networking. We have two primary goals in this evaluation. First, we aim to measure the performance impact of Slipstream for network microbenchmarks and for networked applications. Second, to investigate the performance impacts in more detail, we measure the performance *overheads* incurred by common system calls due to the extra bookkeeping necessary for Slipstream.

We perform all of our experiments on a workstation with a 4-core Intel x86-64 processor with 16GB of DDR3-1333 RAM. This workstation runs Ubuntu 14.04 using stock packages, including most notably Linux 3.13.0-36 as the base kernel, Docker 1.0.1, and OpenJDK 1.7.0_75 for the Java VM. All of the networking configuration parameters for the Linux kernel have been left set to their default values.

5.1 Microbenchmarks

We use the NetPIPE and Netperf microbenchmarks to measure total networking throughput under different networking communication patterns. We use two variants of NetPIPE, one in C and one in Java, to show the impact for two different programming languages; in fact, Slipstream is able to optimize Java code running on OpenJDK as transparently as it does for C code, as explained below. Another microbenchmark, lmbench [20], measures the overhead of Slipstream on common system calls.

5.1.1 lmbench

lmbench [20] is a microbenchmark that measures the overhead of various system calls, which gives an indication of the penalty incurred by using Slipstream in code that may not benefit from its improvements. Selected results are shown in Figure 3; the other numbers are omitted because they are unaffected by Slipstream.

Due to the tracking of file descriptors within userspace, Slipstream naturally adds a small amount of overhead to most system calls that interact with file descriptors. For example, an `open` and `close` pair is 5% slower with Slipstream, while a `select` over 100 socket descriptors is 15% slower.

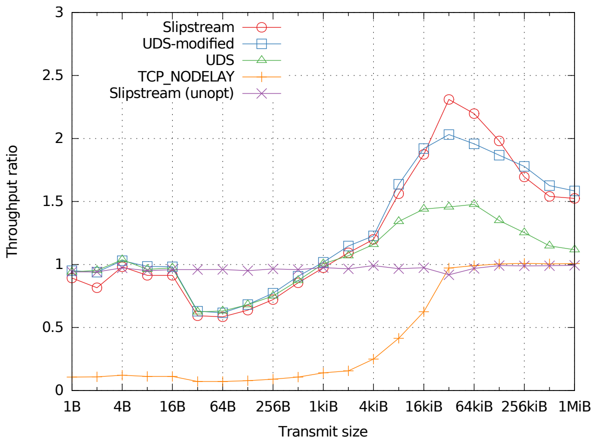
The tracking of file descriptors imposes the largest overheads on `fork` or `exec`. For `fork`, this is due in large part to synchronous communication between `libipc` and `ipcd` that blocks the actual system call. In contrast, the overhead in `exec` is due to loading `libipc` in the memory space of the new process, as well as the overhead of setting up the shared memory object to retain `libipc` state across the call (see Section 3.5).

TCP latency, when the connections have been optimized, is brought down to the same latency as UDS: about 10 microseconds, about half the original TCP latency. However, the initial connection latency is greatly increased due to our need to register the new connection to `ipcd`.

5.1.2 Netperf

Netperf [15] is a microbenchmark to measure total throughput of network connections. Netperf sends data unidirectionally, creating a new socket for each transfer size. The sizes transferred are chosen in logarithmic fashion. Results are presented in Figure 4.

At certain smaller buffer sizes (e.g., 32B-256B), both Slipstream and UDS perform roughly 25-50% worse than the TCP baseline in terms of total throughput. This effect is due to synchronization overhead inside the Linux kernel, which is not observed by the baseline because `TCP_NODELAY` is disabled and TCP buffer coalescing occurs. To validate this, we also compared TCP



Experiment	1 B	32 B	1 KiB	32 KiB	1 MiB
Baseline	2.06	98.28	1625.83	3891.92	5193.10
TCP_NODELAY	0.22	7.11	228.34	3785.77	5232.23
Slipstream	1.84	58.30	1585.19	8988.88	7917.70
Slipstream (unopt)	1.94	94.32	1585.71	3576.18	5159.05
UDS	1.94	61.36	1636.47	5670.26	5801.53
UDS (modified)	1.96	61.93	1655.32	7905.42	8226.89

Figure 4: Throughput as measured by Netperf, with a baseline of TCP without Slipstream or `TCP_NODELAY` specified. The table contains a subset of the throughput results, measured in MB/s.

performance with `TCP_NODELAY` enabled; that curve clearly shows the negative impact of eliminating TCP buffering. At higher data sizes, the relative overhead of the synchronization vs. data transfer is reduced, and Slipstream and UDS sockets both perform better than the baseline, mimicking the results for other benchmarks.

Surprisingly, we observed an increase in throughput with Slipstream compared to using UDS for some of the larger data sizes. On further investigation, we found that this extra speed is primarily due to Netperf using a very small socket receive buffer size (2304 bytes) for the UDS tests. When we changed the Netperf code to not set a socket buffer size (labeled "UDS-modified" in the graph), the apparent effect largely disappears.

Finally, to measure the impact of using Slipstream when optimization is not possible, we ran Netperf using Slipstream only on the client. As shown in the figure, performance was generally very close to that without using Slipstream, on average 3.5% slower than baseline.

5.1.3 NetPIPE

NetPIPE [28] is another microbenchmark that measures the throughput of TCP. NetPIPE differs from Netperf in that it transfers its data bidirectionally and that it reuses the same socket for all of the transfer sizes.

Both variants of NetPIPE use the same basic networking structure, in which the client socket sends data of a given buffer size that the server receives, and then the server sends back that data; the process repeats until sufficient measurements are taken to reliably estimate the throughput. They also both use the idiomatic synchronized socket functions for the language—`send` and `recv` in C, and `java.io.InputStream` and `java.io.OutputStream` in Java.

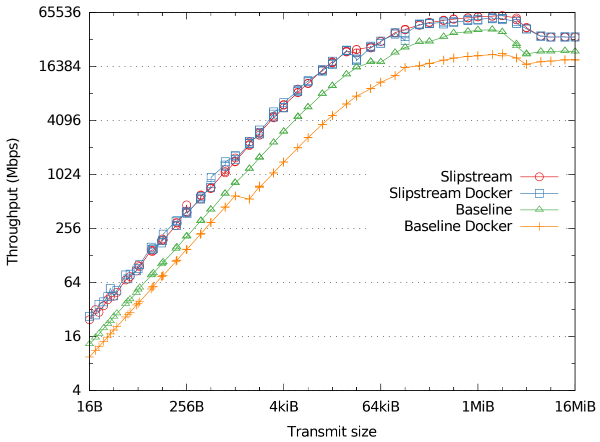


Figure 5: NetPIPE-C performance both with and without Docker containers.

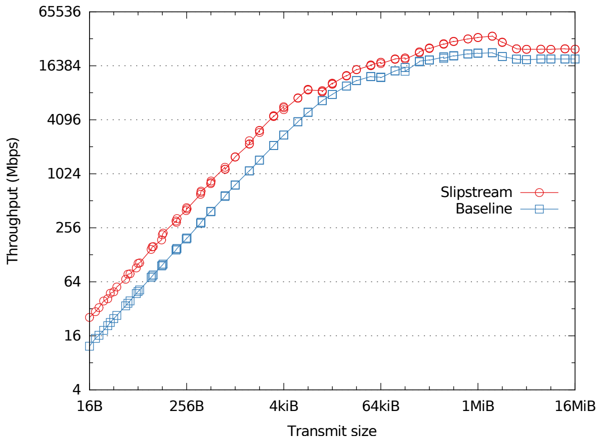


Figure 6: NetPIPE-Java, run only outside of Docker.

The results of running NetPIPE-C are presented in Figure 5, and the results for NetPIPE-Java are presented in Figure 6. For sizes less than about 64KB, Slipstream is able to consistently achieve around 70-150% more throughput compared to baseline TCP. At higher sizes, Slipstream does not provide as much improvement, but is still able to produce at least a 40% increase in throughput.

We also observe that Slipstream optimizes Java networking performance transparently, with no changes to the JVM or the application. Slipstream is able to achieve this because it interposes on `libc`, which is also used by OpenJDK, providing essentially the same benefits to Java programs as to C programs.

5.2 Application Benchmarks

Memcached and PostgreSQL are two example applications that are sometimes used in ways that put the client and server on the same host. We evaluate

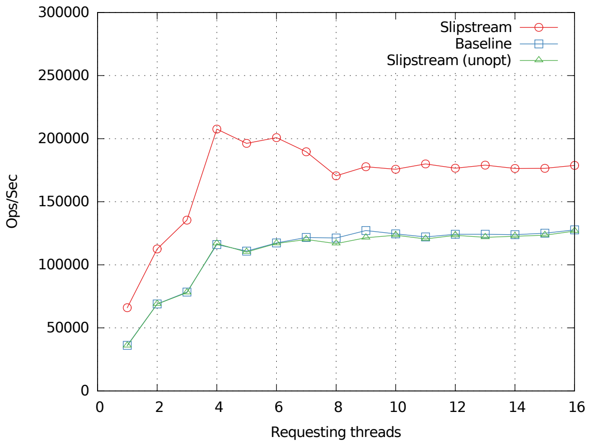


Figure 7: Memcached throughput on host system

these applications with representative workloads to (a) demonstrate that Slipstream is indeed fully transparent for important real applications; and (b) to determine the impact of improving TCP performance for real applications. In addition to measuring performance benchmarks, we used Slipstream on a set of applications—ZeroMQ, OpenSSH, Jenkins (Java), Apache, iperf, simple python TCP client/server, nepim—to informally evaluate compatibility when local to remote TCP communications operate through Slipstream: all of these functioned correctly, and all except OpenSSH were successfully optimized. OpenSSH writes to a socket from multiple processes in a way that we do not currently support in our implementation.

5.2.1 Memcached

Memcached [10] is a distributed, in-memory key-value store that is primarily intended to be used to cache database queries for web applications. While Memcached can be configured to listen on Unix domain sockets instead of TCP, feature requests to allow it to listen on both have been rejected since the particulars of the socket it listens on are used in the distributed hash function [16].

For testing, we run Memcached using a single server and 2GB of storage. Queries are executed against a pool of 10000 items each between 1 byte and 4KB in size. The number of connections is varied, and the average number of operations executed per second is observed. Results are presented in Figure 7. Slipstream provides a 25%-to-100% improvement in throughput, which would be a substantial benefit for small Web sites where most of the traffic is local. When using Slipstream on the client but not the server, we measured on average 1-3% slowdown.

5.2.2 pgbench

pgbench [13] is a benchmark for PostgreSQL, a widely used open-source relational database. We run pgbench with two separate workloads, one based on the industry-standard TPC-B benchmark and the other based on a

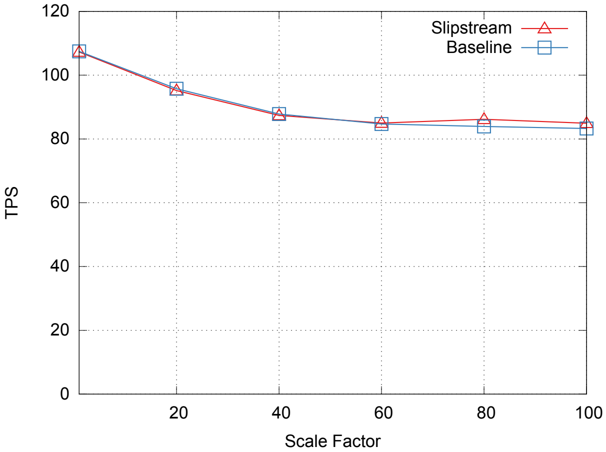


Figure 8: pgbench transactions per second, TPC-B

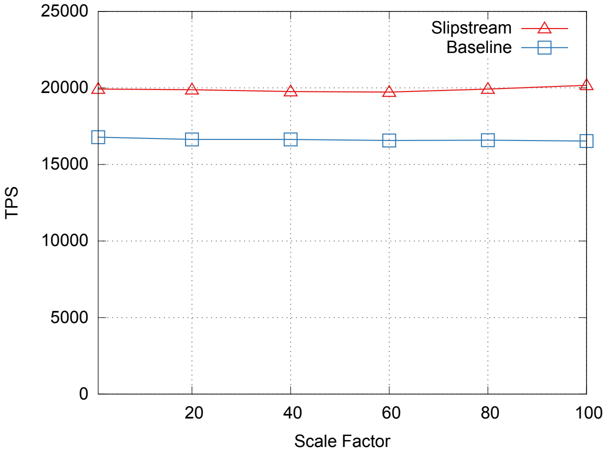


Figure 9: pgbench transactions per second, Select

SELECT-only benchmark, which spends more time in communication. Slipstream successfully optimizes all TCP communication in both cases. A benchmark scale factor, N, creates 100000N rows in the database or about $N \cdot 16$ MB of total database size [26]. Figures 8 and 9 show the results (in database transactions per second, or TPS). Each data point represents the average of multiple runs; the variance observed was negligible.

The TPC-B workload shows little improvement, which is not surprising because the workload is designed to stress the database's internals (primarily disk access) [30]: communication changes have little impact.

In contrast, the SELECT workload shows 16-23% improvement in database throughput for all scale factors. This workload performs simple queries that are processed by the database at a much higher rate and, as a result, benefits significantly from communication optimization.

5.3 Docker

Slipstream is able to detect and optimize local TCP traffic within a single network (i.e., to localhost), but also across virtualized networks executing on the same machine such as those created by Docker. To demonstrate

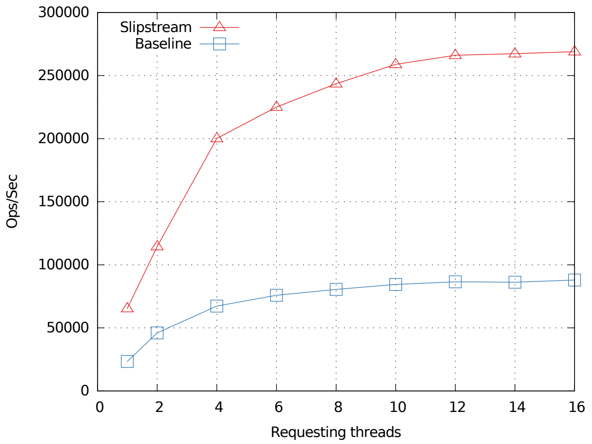


Figure 10: Memcached throughput with Docker

this functionality, and evaluate the performance characteristics of Slipstream in this environment, we conducted additional experiments across Docker containers on the same host. Rerunning our experiments within the same Docker container do not produce different results than by running them on a regular Linux setup.

As different Docker containers have distinct filesystems from the host system, using Slipstream from within a container requires an extra configuration step. If `ipcd` is installed into a Docker container of its own, then the directory containing the UDS that `ipcd` listens on can be also installed on other containers by leveraging the volumes feature of Docker. Alternatively, it is also possible to bind this directory from the host system to a Docker container. Either way, using Slipstream with Docker only requires ensuring that `libipc` is installed within the container and adding a single flag when running the container. *This flexibility is a key benefit of the routing-oblivious design of Slipstream.*

5.3.1 Docker Microbenchmark

We reran our NetPIPE-C microbenchmark using Docker to illustrate the basic performance of Docker networking, also shown in Figure 5. The graph shows that Slipstream is not slowed down when used across Docker containers, whereas normal TCP is, magnifying the TCP throughput improvement to between 150% and 350%. Since Docker containers use separate networking namespaces, the kernel layers need to swap packets between different interfaces, which imposes an extra overhead on TCP transfer, layers which are bypassed by Slipstream. Thus Slipstream's benefits are only enhanced when Docker is in use.

5.3.2 Docker Application Benchmark

In addition to the basic TCP throughput benchmark, we also evaluate the performance of Memcached across Docker containers on the same host. This experiment is identical to the Memcached experiment without containers, with the sole difference that the server and client live in separate containers. These results are shown in Figure 10. Comparing with Figure 7, we again see substantially greater improvements due to Slipstream with Docker than without, ranging from about 100% to 200% speedup. These are very large *application-level* improvements, showing that Slipstream can be a valuable and transparent way to improve the overall performance of services that use Docker containers.

6 Related Work

A number of previous systems aim to optimize local interprocess communication. A brief summary and a feature comparison are presented in Table 1, and are discussed in more detail below.

In-Kernel Solutions Recently, operating systems such as Windows [5], AIX [23], and Solaris [19], have made available localhost TCP optimizations. In general, they all bypass the lower levels of the kernel networking stack, only forming TCP and not IP packets. Performing these optimizations within the existing networking stack simplifies identifying local-only traffic and provides a fast-path for local streams.

Unfortunately, performing these optimizations within the OS has several drawbacks: the implementations are kernel-specific and other systems cannot benefit (e.g., Linux does not support it, although there have been efforts to add it [7]); OS upgrades are often slow to be adopted widely; and applications have no control over whether or not the optimization is available on a given system. In contrast, Slipstream is relatively easy to port, at least across Unix-like systems; it is easy to deploy (e.g., it does not even require superuser privileges to install); and application developers that choose to do so can incorporate the system fairly easily.

VMM solutions for inter-VM communication: Several approaches to improving performance of communication between co-located virtual machines have been described [17, 34, 35], all focusing on Xen. These solve similar communication inefficiencies as Slipstream, but either require application modification [35], guest kernel modification [17, 34, 35], are not fully automatic [17, 35], or operate at the IP layer so TCP overheads are not eliminated [34].

Language-Specific Solutions: The interfaces provided by languages for IPC are often at a much higher level than the basic operations provided by the system. For example, Java Fast Sockets [29] is able to greatly improve communication of Java applications with techniques such as avoiding costly serialization in situations in which the data can be passed through shared memory. While these optimizations are difficult with a language-agnostic solution like Slipstream, Slipstream is able to

Category	Prior Work	Application Transparency[1]	OS Transparency[2]	Sockets[3]	Misc.
OS Impl.	Win. FastPath [5]	✗ Opt-in	✗ Included in OS already	✗	
	Solaris TCP Fusion [19]	✓ Opt-Out	✗ Included in OS already	✓	
	AIX fastlo [23]	✗ Opt-in	✗ Included in OS already	✓	
	Linux TCP Friends [7]	?	✗ Floating Kernel patch	✓	
VM-VM	XWay [17]	✓	✗ Guest kernel patch		
	XenSocket [35]	✗ `AF_XEN`	–	✓	
	XenLoop [34]	✓	✗ Guest kernel module		
User. Stack	mTCP [14]	✗ Similar API	✗ NIC driver for packet library	✗	
	Sandstorm [18]	✗ Specialized for App.	✗ requires netmap [24] kernel support	✗	
User. Shim	Fable [27]	✓ Limited, ns-based	✗ System call for name service	✓	
	Java Fast Sockets [29]	✓ Java.net.Socket	✓	✗	Java-Only
	Universal Fast Sockets [2]	✓	✓	✓	Commercial
	FastSockets [25]	✓	✓ Uses Active Messages	✓	
	Slipstream	✓	✓	✓	

[1] Application Transparency (no application modifications required)

[2] OS Transparency (no OS modifications required, no use of OS-specific tech)

[3] Provides Socket Interface (POSIX, Linux)

Table 1: Prior Work: Categories and Features

optimize applications that use sockets regardless of source language, as our results illustrate for C/C++ and Java.

Transparent Userspace Libraries: The FABLE library, which is only described in a position paper [27], provides automatic transport selection and dynamic reconfiguration of the I/O channel. FABLE provides a socket compatibility layer that uses a new system call for looking up a name mapping (implying that FABLE is not a pure userspace solution) to identify communication with hosts for which it may be able to provide a more efficient transport. Without any information about an implementation, it is unclear how well this compatibility layer works. However, the dynamic switching of transports in Slipstream is very similar to their reconfigurable I/O channels.

Fast Sockets [25] is a userspace library that provides a sockets interface on top of Active Messages [33]. This is superficially similar to Slipstream, but Fast Sockets assumes it can determine which transport to use by inspecting the address, which requires static configuration and a rigid network topology. In contrast, Slipstream focuses on automatic detection of inefficient communication without relying on network topology details and switches transports on-the-fly.

Universal Fast Sockets (UFS)[2] is a commercial solution to optimize local communication transparently. Like Slipstream, UFS uses a shared userspace library to interpose on application activity, but other details of how it operates are proprietary and unclear.

Explicit Userspace Solutions Many software libraries provide *explicit* messaging abstractions for application use [3, 6, 32]. Without the limitations of existing interfaces, impressive performance results are possible, but applications need to be modified to use these frameworks, and seeing the best results may require deep changes to the fundamental structure of an application.

Several researchers have explored moving the network stack out of the operating system and entirely into userspace, citing many performance benefits [14, 22, 24] Userspace networks stacks are components of popular OS designs including the microkernel [11] and exokernel [9] approaches. Some work goes further and collapses the entire network stack into the application [18], providing a specialized stack entirely in userspace. More recent work refactors the OS network stack into a control plane (which stays in the kernel) and separate data planes (which run in protected, library-based operating systems) [4]. All of these efforts redesign the networking stack from the ground up and require kernel modification, application modification, or both. These solutions do not directly aim to optimize local communication, but similar to the in-kernel approaches described above this could likely be added in a straightforward if not portable manner.

7 Acknowledgements

This work was supported in part by the Office of Naval Research grant numbers N00014-12-1-0552 and N00014-4-1-0525, and by the AFOSR under MURI award FA9550-09-1-0539.

8 Conclusion

Slipstream is a novel system for the optimization of TCP communication that requires neither OS nor application modification, which allows it to be easily and rapidly deployed. Our evaluations show that our system is capable of achieving significant performance benefits, at least 16% more throughput than TCP, and up to 200% if Docker is involved, both on real applications in real usage scenarios. Slipstream's minimal assumptions allow it to be used in a variety of network topologies and to use a variety of faster local transports, capabilities we plan to explore in future work. We believe that Slipstream provides an excellent base for reducing the overhead of IPC in applications that is usable across a wide variety of applications and setups.

References

[1] HAProxy. http://www.haproxy.org.

[2] Universal Fast Sockets. http://torusware.com/product/universal-fast-sockets-ufs/.

[3] ZeroMQ. http://zeromq.org.

[4] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 49–65.

[5] BRIGGS, E. Fast TCP loopback performance and low latency with Windows Server 2012 TCP Loopback Fast Path. http://blogs.technet.com/b/wincat/archive/2012/12/05/fast-tcp-loopback-performance-and-low-latency-with-windows-server-2012-tcp-loopback-fast-path.aspx, Dec. 2012.

[6] CAMERON, D., AND REGNIER, G. *The virtual interface architecture*. Intel Pr, 2002.

[7] CORBET, J. TCP friends. http://lwn.net/Articles/511254/, Aug. 2012.

[8] DORAN, T. Building a smarter application stack, 2014. Presented at DockerCon.

[9] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266.

[10] FITZPATRICK, B. Distributed caching with Memcached. *Linux journal 2004*, 124 (2004), 5.

[11] GOLUB, D. B., JULIN, D. P., RASHID, R. F., DRAVES, R. P., DEAN, R. W., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., AND BOHMAN, D. Microkernel operating system architecture and Mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (1992), pp. 11–30.

[12] INTERNET ASSIGNED NUMBERS AUTHORITY. Service Name and Transport Protocol Port Number Registry, May 2015. (Date last updated.).

[13] ISHII, T. pgbench. http://www.postgresql.org/docs/devel/static/pgbench.html.

[14] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 489–502.

[15] JONES, R. The Netperf benchmark. http://www.netperf.org.

[16] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[17] KIM, K., KIM, C., JUNG, S.-I., SHIN, H.-S., AND KIM, J.-S. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 11–20.

[18] MARINOS, I., WATSON, R. N. M., AND HANDLEY, M. Network stack specialization for performance. *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks - HotNets-XII* (2013), 1–7.

[19] MAURO, J., AND MCDOUGALL, R. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[20] MCVOY, L., AND STAELIN, C. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), ATEC '96, USENIX Association, pp. 23–23.

[21] MERKEL, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[22] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 337–350.

[23] QUINTERO, D., CHABROLLES, S., CHEN, C., DHANDAPANI, M., HOLLOWAY, T., JADHAV, C., KIM, S., KURIAN, S., RAJ, B., RESENDE, R., ET AL. *IBM Power Systems Performance Guide: Implementing and Optimizing*. IBM Redbooks, 2013.

[24] RIZZO, L. netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 9–9.

[25] RODRIGUES, S. H., ANDERSON, T. E., AND CULLER, D. E. High-performance local area communication with Fast Sockets. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1997), ATEC '97, USENIX Association, pp. 20–20.

[26] SMITH, G. Introduction to pgbench. http://www.westnet.com/~gsmith/content/postgresql/pgbench-intro.pdf, 2009. Presented at PG East.

[27] SMITH, S., MADHAVAPEDDY, A., SMOWTON, C., SCHWARZKOPF, M., MORTIER, R., WATSON, R. M., AND HAND, S. The case for reconfigurable I/O channels. In *RESoLVE workshop at ASPLOS* (2012), vol. 12.

[28] SNELL, Q. O., MIKLER, A. R., AND GUSTAFSON, J. L. NetPIPE: A network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems* (1996), vol. 6, Washington, DC, USA).

[29] TABOADA, G. L., TOURIÑO, J., AND DOALLO, R. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun. 31*, 17 (Nov. 2008), 4049–4059.

[30] TÖZÜN, P., PANDIS, I., KAYNAK, C., JEVDJIC, D., AND AILAMAKI, A. From a to e: Analyzing TPC's OLTP benchmarks: The obsolete, the ubiquitous, the unexplored. In *Proceedings of the 16th International Conference on Extending Database Technology* (New York, NY, USA, 2013), EDBT '13, ACM, pp. 17–28.

[31] VASSERMAN, E. Y., HOPPER, N., AND TYRA, J. SilentKnock: practical, provably undetectable authentication. *International Journal of Information Security 8*, 2 (2009), 121–135.

[32] VINOSKI, S. Advanced message queuing protocol. *IEEE Internet Computing 10*, 6 (Nov. 2006), 87–89.

[33] VON EICKEN, T., CULLER, D. E., GOLDSTEIN, S. C., AND SCHAUSER, K. E. Active Messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1992), ISCA '92, ACM, pp. 256–266.

[34] WANG, J., WRIGHT, K.-L., AND GOPALAN, K. XenLoop: A transparent high performance inter-VM network loopback. *Cluster Computing 12*, 2 (June 2009), 141–152.

[35] ZHANG, X., MCINTOSH, S., ROHATGI, P., AND GRIFFIN, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware* (New York, NY, USA, 2007), Middleware '07, Springer-Verlag New York, Inc., pp. 184–203.