University of Glasgow

School of Computing Science

Network Programming Interfaces

Advanced Operating Systems (M)

Lecture 7

Colin Perkins | https://csperkins.org/ | Copyright © 2018 | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

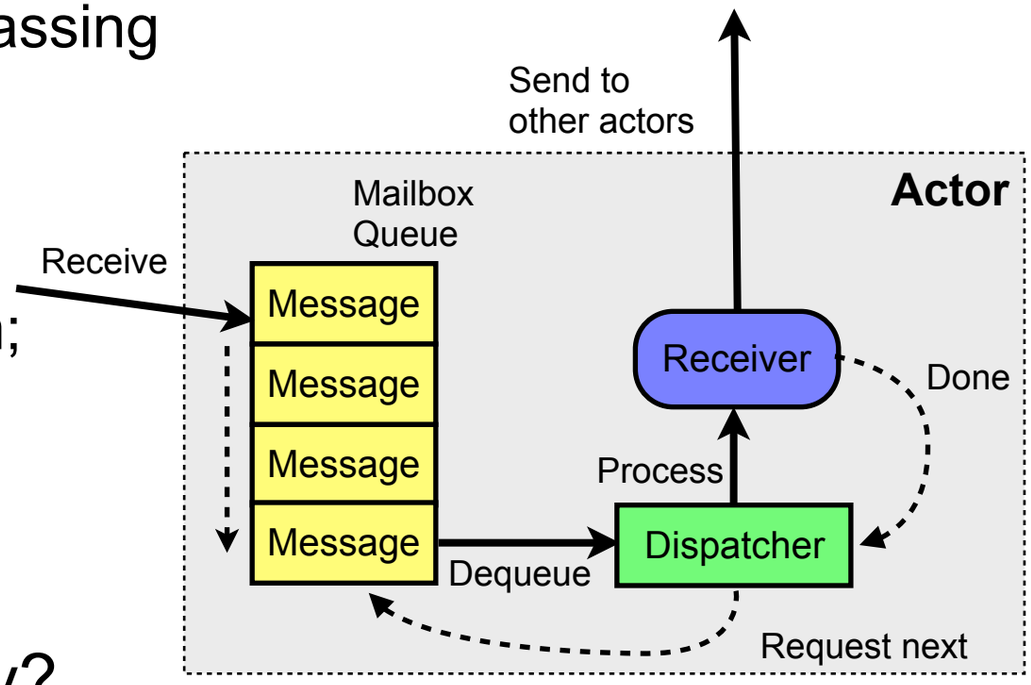
- Higher level network programming abstractions
  - Actors, sockets, and network protocols
  - Asynchronous I/O frameworks
- High performance networking

CC BY ND

Higher level network programming abstractions

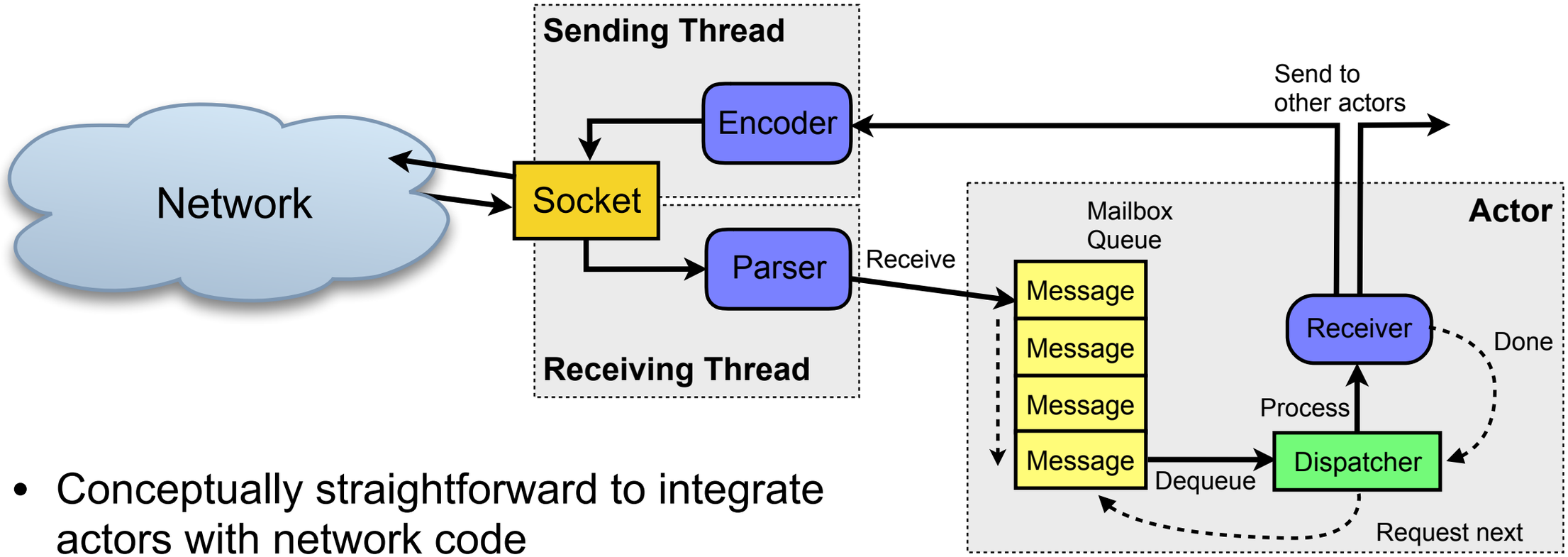
Message Passing and Network Protocols

- Recap:
  - Actor-based framework for message passing
  - Each actor has a receive loop
  - Calls to one function per state
  - Messages delivered by runtime system; processed sequentially
  - Actor can send messages in reply; return identity of next state
- Can we write network code this way?
  - Send data by sending a message to an actor representing a socket
  - Receive messages representing data received on a socket



Colin Perkins | https://csperkins.org/ | Copyright © 2018

Integrating Actors and Sockets



- Conceptually straightforward to integrate actors with network code
  - Runtime system maintains sending and receiving threads that manage the sockets
  - Receiving thread reads network protocol data from sockets, parses into messages, dispatches into the actor runtime
  - Sending thread receives messages from the actor runtime, encodes them as formatted packets of the network protocol, and sends via appropriate socket

- Network communication integrated with the actor system as message passing
- Adapters format and parse messages into the wire format

CC BY ND

Parsing and Serialisation

- Parsing and serialisation requires socket awareness of the protocol:
  - E.g., make `Socket` an abstract class with `encode` and `parse` methods that are implemented by concrete subclasses and understand particular protocol
  - The `encode` method takes an object representing a protocol message, and returns a byte array that can be sent via the socket
  - The `parse` method takes a byte array received from the socket, and returns an object representing the protocol message
- Encapsulates parsing and serialisation – everything outside the socket uses strongly typed objects, not raw protocol data

BY ND

Actors for Network Programming

- Conceptually clean – the simple message passing model abstracts away details of network transport
  - Easy to program and debug
  - Encapsulates parsing and serialisation
  - Helps correctness – on-the-wire format hidden, applications operate on strongly typed objects:

```
def receive = {
  case HttpRequest(GET, Uri.Path("/ping"), _, _, _) => {
    sender ! HttpResponse(entity = “PONG”)
  }
  case HttpRequest(PUT, …) => {
    …
  }
  case _: Tcp.ConnectionClosed => {
    …
  }
}
```

(Fragment of Scala+Akka+Spray code)

CC BY ND

Actors for Network Programming

- Performance reasonable, but likely not outstanding
- Conceptually well suited to implementing protocols
  - Asynchronous reception matches network behaviour
  - Operates at high level of abstraction
  - Examples: EJabberD, WhatsApp → Erlang
- Potentially ill-suited to applications that track details of network behaviour
  - The “send message and forget it” style isn’t well suited to monitoring the progress or timing of delivery (e.g., for an adaptive video conference app)

CC BY ND

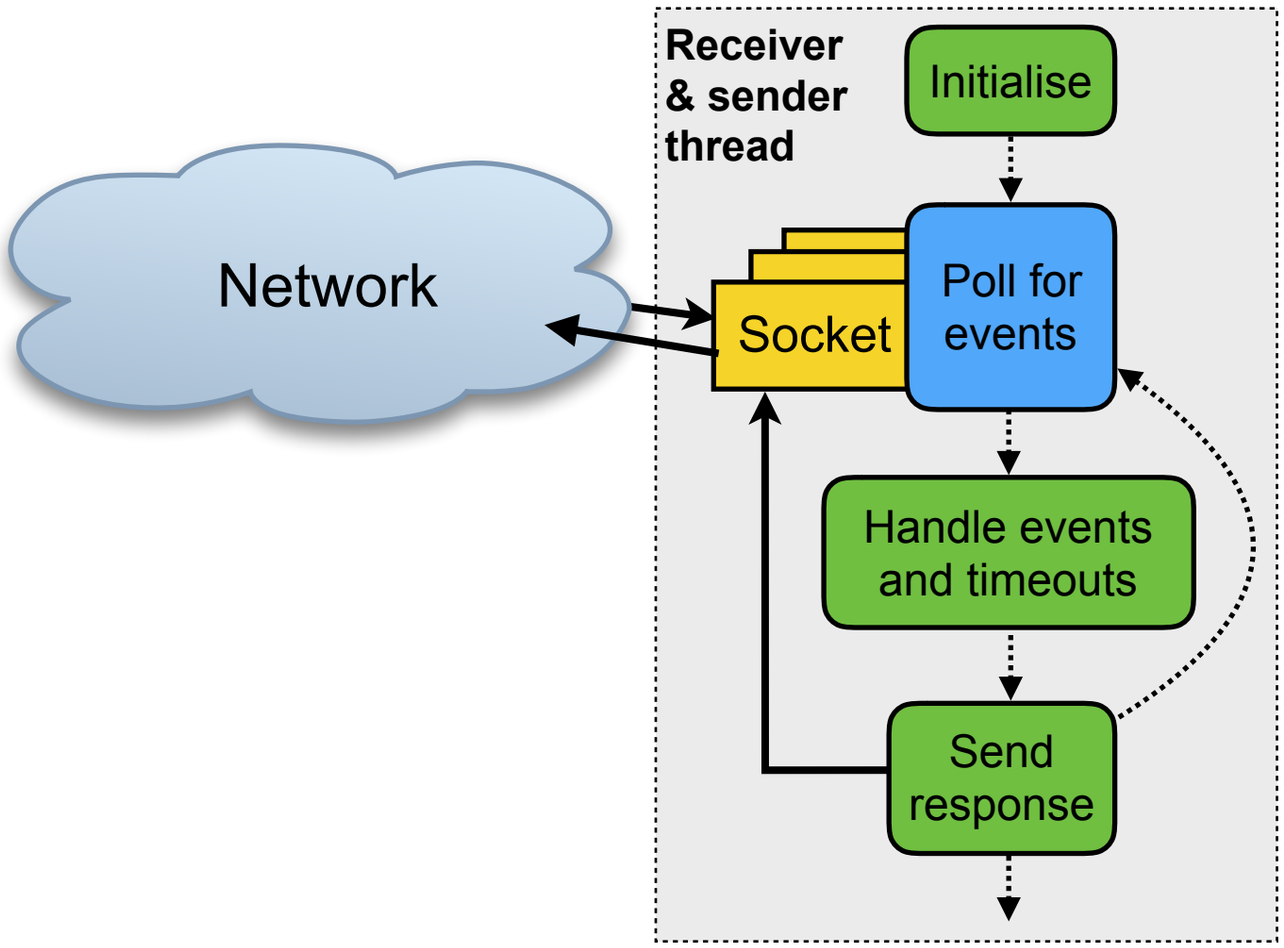
Asynchronous I/O Frameworks

- Key insight from actor-based I/O: packet reception should be asynchronous to match the network
- However…
  - Sending API should be synchronous
    - The back pressure of a blocking send is useful to avoid congestion/sending queue build-up
    - Applications that query the state of a network interface need to know what is happening now, not what happened some time in the past, after a message exchange
    - Much simpler to implement in a synchronous manner
  - Actor-based implementations have relatively high overhead

- Implement as blocking event loop, with timeout – an asynchronous (i.e., event driven) I/O framework

CC BY ND

Event Loops



- Many implementations with broadly similar functionality:
  - `select()` and `poll()`: portable, but slow with large numbers of sockets
  - `epool()` (Linux), `kqueue` (FreeBSD/ macOS, iOS): efficient, less portable
  - `libevent`/`libev`/`libuv` wrap the above in a unified API (`libuv` currently seems most popular)
- Basic operation:
  - Pass a set of sockets, file descriptors, etc., to the kernel
  - Poll those for events – will block until something happens or timeout occurs
  - Process events, send any responses, loop back for next event
- Single threaded event loop
  - Long-running operations are passed to separate worker threads

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Example: libev

…

```
static void udp_cb(EV_P_ ev_io *w, int revents) {
  // Called with data is readable on the UDP socket
}
```

Callback to handle event and send response if needed

```
int main(void) {
    int port = DEFAULT_PORT;
    puts("udp_echo server started...");

    // Setup a udp listening socket.
    sd = socket(PF_INET, SOCK_DGRAM, 0);
    bzero(&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr*) &addr, sizeof(addr)) != 0)
        perror("bind");

    // Do the libev stuff.
    struct ev_loop *loop = ev_default_loop(0);
    ev_io udp_watcher;
    ev_io_init(&udp_watcher, udp_cb, sd, EV_READ);
    ev_io_start(loop, &udp_watcher);
    ev_loop(loop, 0);

    close(sd);
    return EXIT_SUCCESS;
}
```

Initialise

Loop, processing events

BY ND

Example: Rust MIO

```
fn main() {
    let ip       = IpAddr::V4(Ipv4Addr::new(0,0,0,0));
    let port     = 2223;
    let sockaddr = SocketAddr::new(ip, port);
    let socket   = UdpSocket::bind(&sockaddr).unwrap();

    let poll = Poll::new().unwrap();
    let mut events = Events::with_capacity(1024);

    poll.register(&socket, TOKEN, Ready::readable(), PollOpt::edge()).unwrap();

    loop {
        poll.poll(&mut events, None).unwrap();

        for event in &events {
            match event.token() {
                TOKEN => {
                    println!("got event");
                }
                _ => panic!("event with unexpected token")
            }
        }
    }
}
```

Initialise

Loop, processing events

Handle events

CC BY ND

Event Loops

- Asynchronous I/O via event loops – efficient and highly scalable
  - Some implementations callback based (e.g., `libuv`), others expose the event set and allow applications to dispatch (e.g., Rust MIO)
    - I tend to prefer the latter, pattern matching, style – code is more obvious
  - Provided it does no long-running calculations, a single asynchronous I/O thread can usually saturate a single network interface
    - Use one thread per network interface
    - Use a thread pool to handle long-running calculations, passing requests to it for processing
- Model is the same as actor-based systems:
  - Sequence of events (i.e., messages) delivered to application; process each in turn, sending replies as needed
  - However, have direct access to underlying socket for sending operations, monitoring performance

CC BY ND

Higher-level Abstractions

- Event-driven asynchronous I/O efficient and flexible → but can lead to obfuscated code
  - Implementation split across numerous callbacks; hard to follow the logic
  - Flexible, to implement complex protocols
- Can we provide an abstraction to simplify common protocol styles?
  - e.g., a server that responds to a single request with a single response

CC BY ND

Your Server as a Function (1)

- Conceptually, a server represents a function:

```
fn service_request(&self, req : Request) -> Future<Response, Error>;
```

- Takes a request and returns a *future* – an eventual response or error:

```
enum Future<Response, Error> {
  InProgress,
  Success<Response>,
  Failed<Error>
}
```

- Combine with parsing and encoding functions:

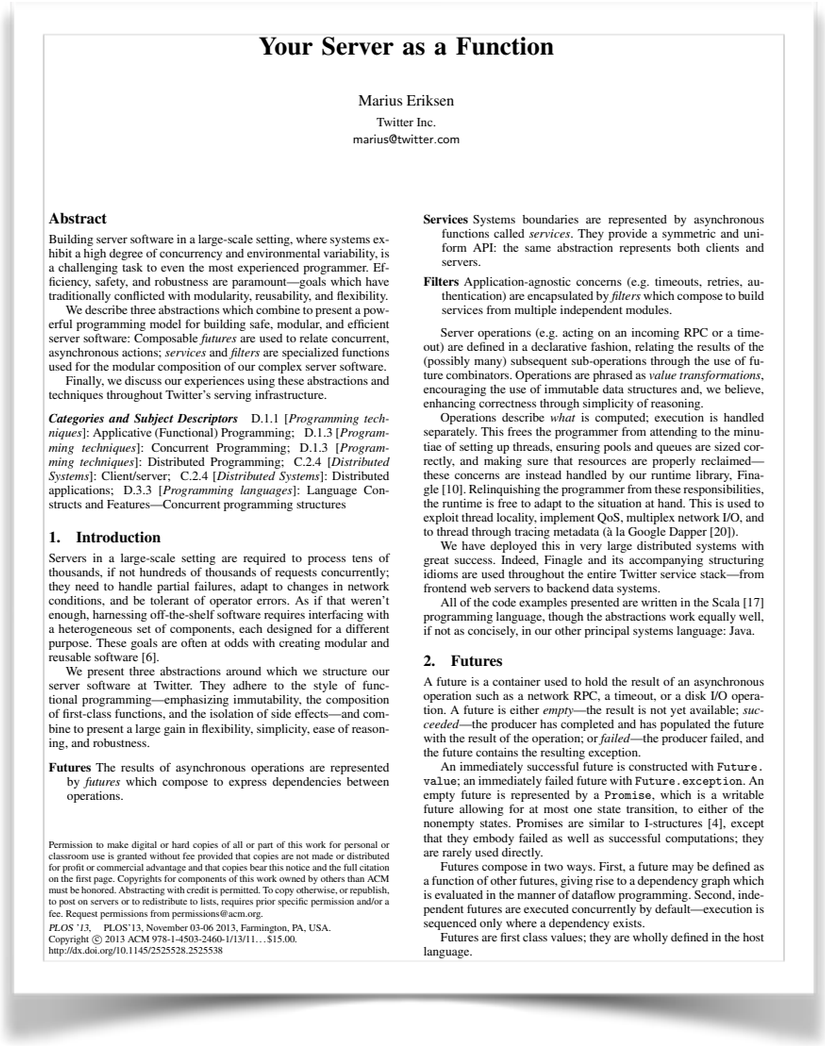
```
fn parse(&self, data : &[u8]) -> Result<Request, Error>;
fn encode(&self, res : Response) -> [u8];
```

Convert raw protocol data to strongly-typed objects

BY ND

Your Server as a Function (2)

- These three functions, plus definitions of `Request`, `Response`, and `Error` types can specify a server
- Plug into a library handling asynchronous I/O operations/scheduling calls to the functions
- Potentially allows writing services at a very high level of abstraction



Your Server as a Function

Marius Eriksen
Twitter Inc.
marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a powerful programming model for building safe, modular, and efficient server software: Composable *futures* are used to relate concurrent, asynchronous actions; *services* and *filters* are specialized functions used for the modular composition of our complex server software.

Finally, we discuss our experiences using these abstractions and techniques throughout Twitter's serving infrastructure.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Concurrent Programming; D.1.3 [*Programming techniques*]: Distributed Programming; C.2.4 [*Distributed Systems*]: Client/server; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures

1. Introduction

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Futures The results of asynchronous operations are represented by *futures* which compose to express dependencies between operations.

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Server operations (e.g. acting on an incoming RPC or a timeout) are defined in a declarative fashion, relating the results of the (possibly many) subsequent sub-operations through the use of future combinators. Operations are phrased as *value transformations*, encouraging the use of immutable data structures and, we believe, enhancing correctness through simplicity of reasoning.

Operations describe *what* is computed; execution is handled separately. This frees the programmer from attending to the minutiae of setting up threads, ensuring pools and queues are sized correctly, and making sure that resources are properly reclaimed—these concerns are instead handled by our runtime library, Finagle [10]. Relinquishing the programmer from these responsibilities, the runtime is free to adapt to the situation at hand. This is used to exploit thread locality, implement QoS, multiplex network I/O, and to thread through tracing metadata (à la Google Dapper [20]).

We have deployed this in very large distributed systems with great success. Indeed, Finagle and its accompanying structuring idioms are used throughout the entire Twitter service stack—from frontend web servers to backend data systems.

All of the code examples presented are written in the Scala [17] programming language, though the abstractions work equally well, if not as concisely, in our other principal systems language: Java.

2. Futures

A future is a container used to hold the result of an asynchronous operation such as a network RPC, a timeout, or a disk I/O operation. A future is either *empty*—the result is not yet available; *succeeded*—the producer has completed and has populated the future with the result of the operation; or *failed*—the producer failed, and the future contains the resulting exception.

An immediately successful future is constructed with `Future.value`; an immediately failed future with `Future.exception`. An empty future is represented by a `Promise`, which is a writable future allowing for at most one state transition, to either of the nonempty states. Promises are similar to I-structures [4], except that they embody failed as well as successful computations; they are rarely used directly.

Futures compose in two ways. First, a future may be defined as a function of other futures, giving rise to a dependency graph which is evaluated in the manner of dataflow programming. Second, independent futures are executed concurrently by default—execution is sequenced only where a dependency exists.

Futures are first class values; they are wholly defined in the host language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLOS '13, PLOS'13, November 03-06 2013, Farmington, PA, USA.
Copyright © 2013 ACM 978-1-4503-2460-1/13/11…$15.00.
http://dx.doi.org/10.1145/2525528.2525538

- Examples:
  - Twitter's `Finagle` library in Scala
  - The `Tokio` framework for Rust
- Background reading (examinable, but will not be discussed in tutorial):
  - M. Eriksen, "Your server as a function", Proceedings of Workshop on Programming Languages and Operating Systems, Farmington, PA, USA, November 2013. ACM. DOI: 10.1145/2525528.2525538

CC BY ND

Summary

- Many types of network code are a good fit for actor frameworks
  - Protocol messages parsed/encoded to/from Actor messages
  - But – relatively high overheads; poorly suited to low-level applications
- Event driven asynchronous I/O framework solve many of these issues – but can obfuscate code
- Event driven approaches based on pattern matching lead to more obvious code than callback based systems
- The server-as-a-function approach can abstract asynchronous I/O into simple, high-level, frameworks for some applications

CC BY ND

High Performance Networking

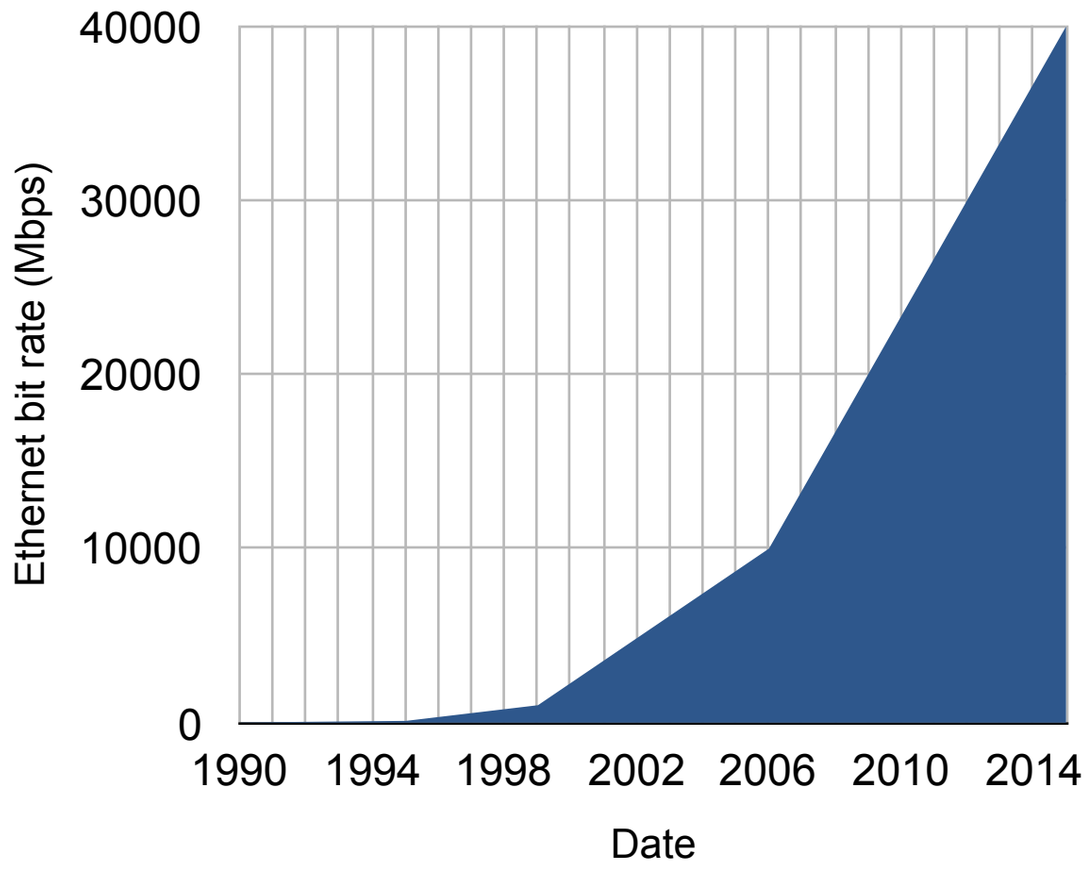
High Performance Networking

- Limitations of the kernel protocol stack
- Alternative network stacks
- Accelerating TCP via API improvements
- Some cautionary remarks

BY ND

Network Performance Growth

- Network performance is increasing faster than CPU performance
  - Chart shows Ethernet bit rate over time – wireless links follow a similar curve
  - Closely tracking exponential growth over time – unlike CPU speed, which stopped growing significantly mid-2000s
- MTU remains constant → packet rate increases
  - Maximum 14,880,952 packets/second on 10Gbps Ethernet (scales linearly with link rate)
  - Minimum size packet is 60 bytes data, with 8 byte preamble, 4 byte CRC; 12 byte inter-frame gap (silent period) between packets
- CPU cycles available to process each packet decreasing



CC BY ND

Limitations of the Kernel Protocol Stack

- Why does the traditional kernel protocol stack offer sub-optimal performance?
- Designed when CPUs were faster than networks
  - Allocates memory for buffers on per-packet basis
  - Copies data multiple times, from NIC (“network interface card”) to kernel to application
  - System call to send/receive each packet
    - Layered architecture offers clean design, but not efficient packet processing
    - Example on right: timing of a sendto() system call on FreeBSD: 950ns total; overheads at each layer boundary due to system call, copies, etc.
- How to redesign the protocol stack to reduce overheads?

File	Function/description	time ns	delta ns
user program	`sendto` system call	8	96
uipc_syscalls.c	`sys_sendto`	104	
uipc_syscalls.c	`sendit`	111	
uipc_syscalls.c	`kern_sendit`	118	
uipc_socket.c	`sosend`	—	
uipc_socket.c	`sosend_dgram` sockbuf locking, mbuf allocation, copyin	146	137
udp_usrreq.c	`udp_send`	273	
udp_usrreq.c	`udp_output`	273	57
ip_output.c	`ip_output` route lookup, ip header setup	330	198
if_ethersubr.c	`ether_output` MAC header lookup and copy, loopback	528	162
if_ethersubr.c	`ether_output_frame`	690	
ixgbe.c	`ixgbe_mq_start`	698	
ixgbe.c	`ixgbe_mq_start_locked`	720	
ixgbe.c	`ixgbe_xmit` mbuf mangling, device programming	730	220
–	on wire	950	

Source: L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012.

CC BY ND

An Alternative Network Stack: netmap

- Changes network API – a mechanism to give an application dedicated control of a NIC
  - A replacement for `libpcap`, not the Sockets API
  - Basis for fast packet capture applications; software router; user-space protocol stack – not general purpose
- Pre-allocate buffers, that are shared between OS and user application; coordinate buffer ownership
  - No memory allocation at time the packets are sent/received
  - No data copies – DMA direct to application accessible memory
  - Fewer system calls – one system call can transfer ownership of multiple buffers between application and kernel
- NIC uses efficient DMA to transfer packets to and from memory; kernel manages synchronisation and memory protection
  - Kernel is the control plane
  - NIC and DMA transfer manage the data plane



netmap: a novel framework for fast packet I/O

Luigi Rizzo, *Università di Pisa, Italy*[*]
Proceedings of the 2012 USENIX Annual Technical Conference, June 2012.
`https://www.usenix.org/conference/atc12/` [†]

Abstract

Many applications (routers, traffic monitors, firewalls, etc.) need to send and receive packets at line rate even on very fast links. In this paper we present *netmap*, a novel framework that enables commodity operating systems to handle the millions of packets per seconds traversing 1..10 Gbit/s links, without requiring custom hardware or changes to applications.

In building *netmap*, we identified and successfully reduced or removed three main packet processing costs: per-packet dynamic memory allocations, removed by preallocating resources; system call overheads, amortized over large batches; and memory copies, eliminated by sharing buffers and metadata between kernel and userspace, while still protecting access to device registers and other kernel memory areas. Separately, some of these techniques have been used in the past. The novelty in our proposal is not only that we exceed the performance of most of previous work, but also that we provide an architecture that is tightly integrated with existing operating system primitives, not tied to specific hardware, and easy to use and maintain.

netmap has been implemented in FreeBSD and Linux for several 1 and 10 Gbit/s network adapters. In our prototype, a single core running at 900 MHz can send or receive 14.88 Mpps (the peak packet rate on 10 Gbit/s links). This is more than 20 times faster than conventional APIs. Large speedups (5x and more) are also achieved on user-space Click and other packet forwarding applications using a libpcap emulation library running on top of *netmap*.

[*]This work was funded by the EU FP7 project CHANGE (257422).
[†]USENIX plans to publish this paper on the *Proceedings of the 2012 USENIX Annual Technical Conference*, which will be available at this URL after June 13, 2012. You may also find this paper, with related material, on the author's site, `http://info.iet.unipi.it/ luigi/netmap/`

1 Introduction

General purpose OSes provide a rich and flexible environment for running, among others, many packet processing and network monitoring and testing tasks. The high rate raw packet I/O required by these applications is not the intended target of general purpose OSes. Raw sockets, the Berkeley Packet Filter [14] (BPF), the AF_SOCKET family, and equivalent APIs have been used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is inadequate for the millions of packets per second (*pps*) that can be present on 1..10 Gbit/s links. In search of better performance, some systems (see Section 3) either run completely in the kernel, or bypass the device driver and the entire network stack by exposing the NIC's data structures to user space applications. Efficient as they may be, many of these approaches depend on specific hardware features, give unprotected access to hardware, or are poorly integrated with the existing OS primitives.

The *netmap* framework presented in this paper combines and extends some of the ideas presented in the past trying to address their shortcomings. Besides giving huge speed improvements, *netmap* does not depend on specific hardware[1], has been fully integrated in FreeBSD and Linux with minimal modifications, and supports unmodified libpcap clients through a compatibility library.

One metric to evaluate our framework is performance: in our implementation, moving one packet between the wire and the userspace application has an amortized cost of less than 70 CPU clock cycles, which is at least one order of magnitude faster than standard APIs. In other words, a single core running at 900 MHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link. The same core running at 150 MHz is well above the capacity

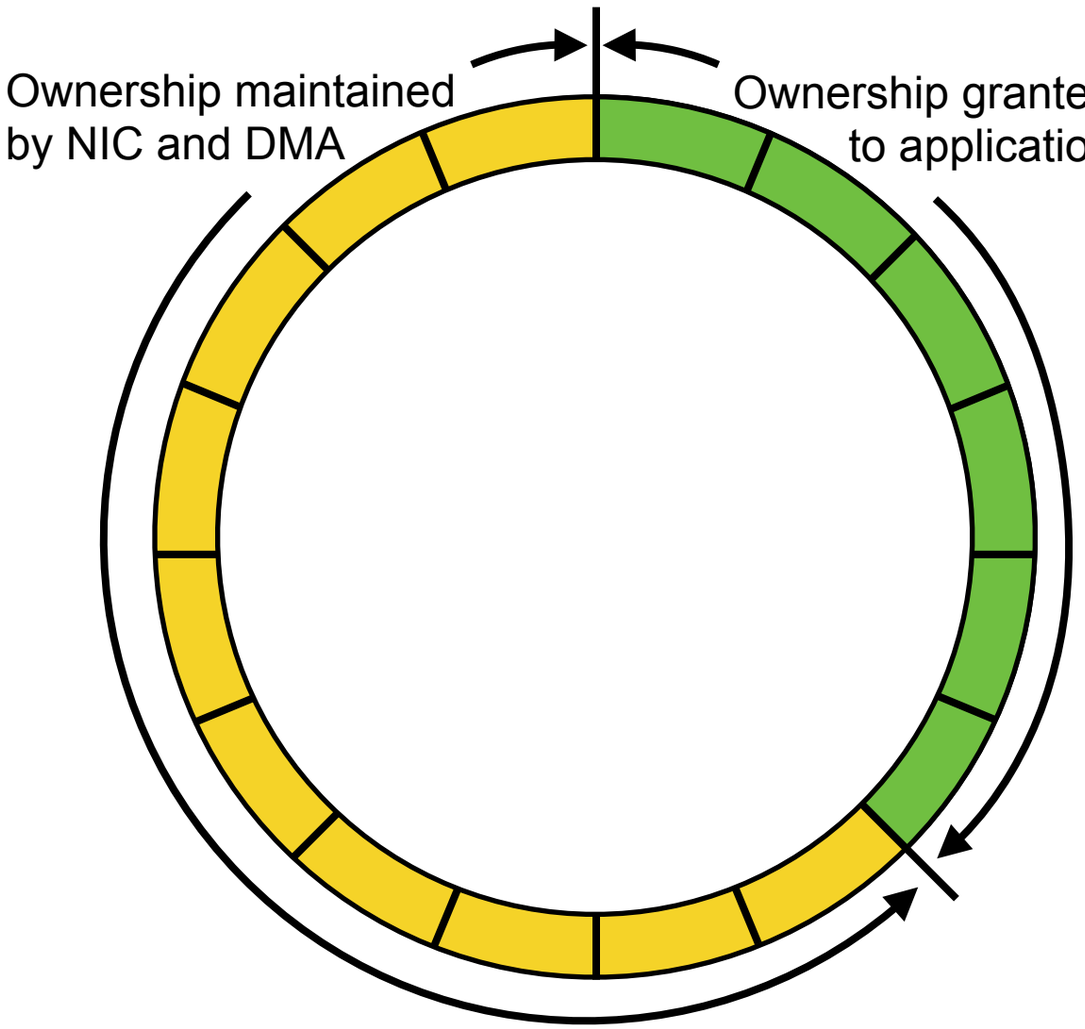
[1]*netmap* can give isolation even without hardware mechanisms such as IOMMU or VMDq, and is orthogonal to hardware offloading and virtualization mechanisms (checksum, TSO, LRO, VMDc, etc.)

1

L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo (paper and video of presentation)

CC BY ND

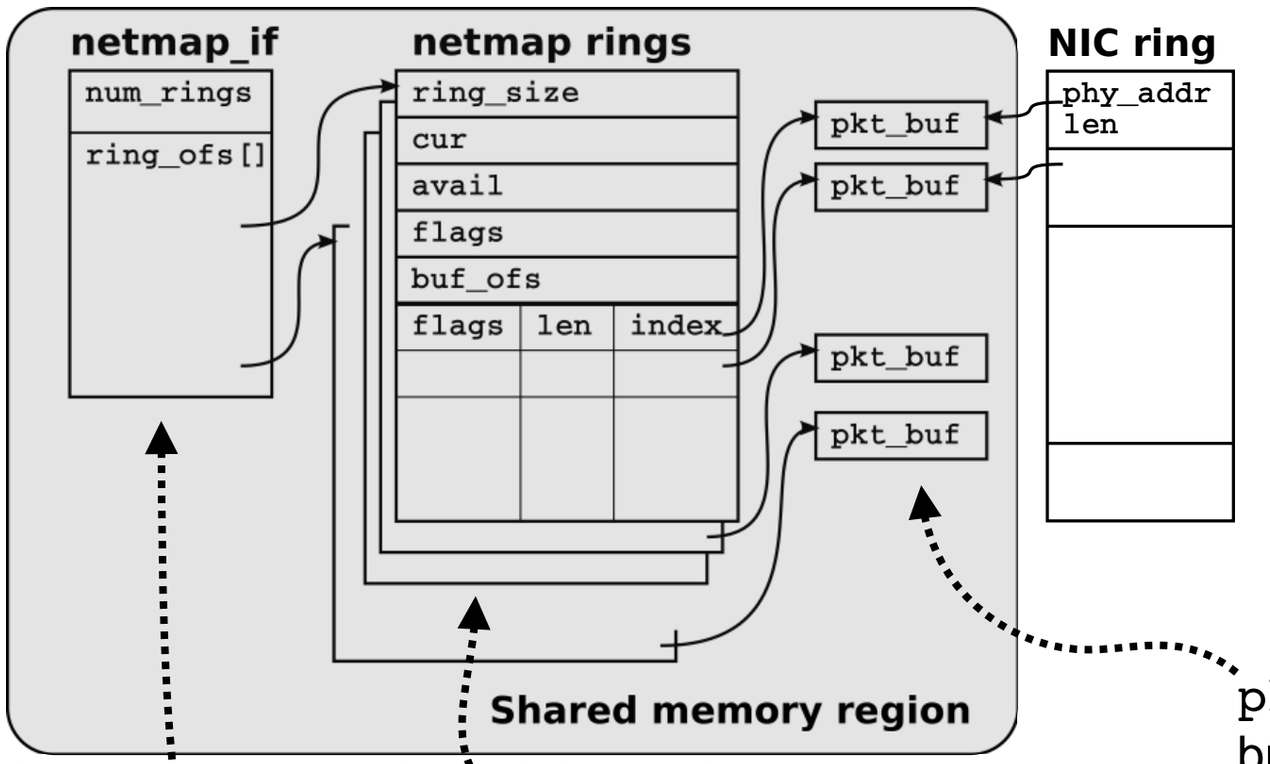
Shared Ownership of NIC Buffers (1)



- Modern NICs maintain circular buffers sized to hold queues of full size packets
  - NIC writes incoming packets direct to one segment of ring buffer via DMA
  - Operating system copies from other segment into lower layer of protocol stack – first of several copies
  - If using netmap → OS disconnected; ownership of ring buffer segment is temporarily granted to application to process packets in place
  - (Analogous for outgoing packets)

Colin Perkins | https://csperkins.org/ | Copyright © 2018

Shared Ownership of NIC Buffers (2)



Source: L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012.

`pkt_buf` objects form the circular buffer, and are in memory shared between NIC and application

`netmap_ring` provides index into the circular buffer; tracks current ownership of each `pkt_buf`

`netmap_if` provides metadata about the interface

CC BY ND

Sample netmap Code

```
struct netmap_if *nifp;
struct nmreq req;
int i, len;
char *buf;

fd = open("/dev/netmap", 0);
strcpy(req.nr_name, "ix0"); // register the interface
ioctl(fd, NIOCREG, &req); // offset of the structure
mem = mmap(NULL, req.nr_memsize, PROT_READ|PROT_WRITE, 0, fd, 0);
nifp = NETMAP_IF(mem, req.nr_offset);
for (;;) {
    struct pollfd x[1];
    struct netmap_ring *ring = NETMAP_RX_RING(nifp, 0);

    x[0].fd = fd;
    x[0].events = POLLIN;
    poll(x, 1, 1000);
    for ( ; ring->avail > 0 ; ring->avail--) {
        i = ring->cur;
        buf = NETMAP_BUF(ring, i);
        use_data(buf, ring->slot[i].len);
        ring->cur = NETMAP_NEXT(ring, i);
    }
}
```

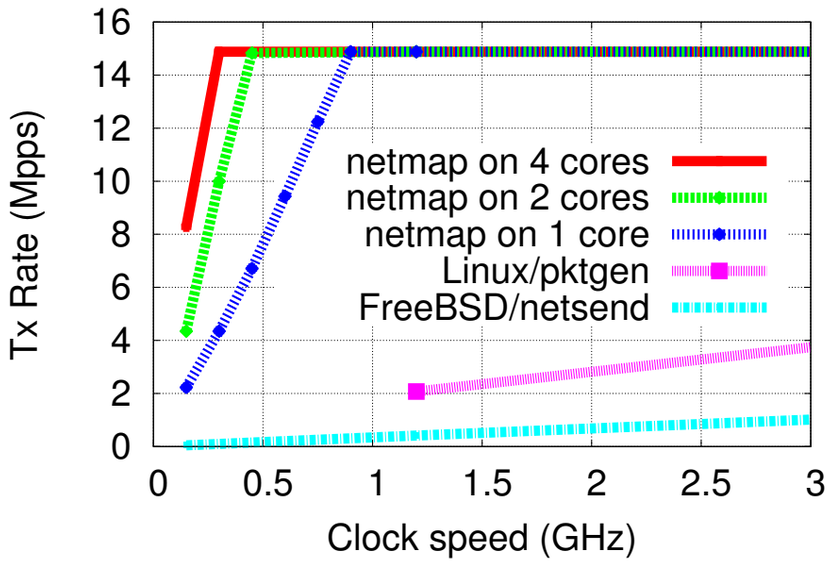
Updates `netmap_ring` structure, based on the received data (`ring->cur` and `ring->avail` only, no data copied, no synchronisation)

Gets pointer to shared `pkf_buf`

CC BY ND

Benefits and Limitations of netmap

- Memory shared between NIC and application
  - Misbehaving applications can read memory owned by NIC – see unpredictable contents since DMA active in this region
  - Kernel data structures are protected – cannot crash the kernel or see other kernel data
- Operates on granularity of network interface
  - Application has access to all traffic on netmap interface
  - Limited applicability – not a replacement for the TCP/IP stack, but well suited to network monitoring or software router implementation
- Performance excellent – saturates 10Gbps Ethernet sending minimum size packets on 900MHz CPU
  - Using minimum size packets requires highest packet rate → highest overhead
  - Similar performance receiving packets

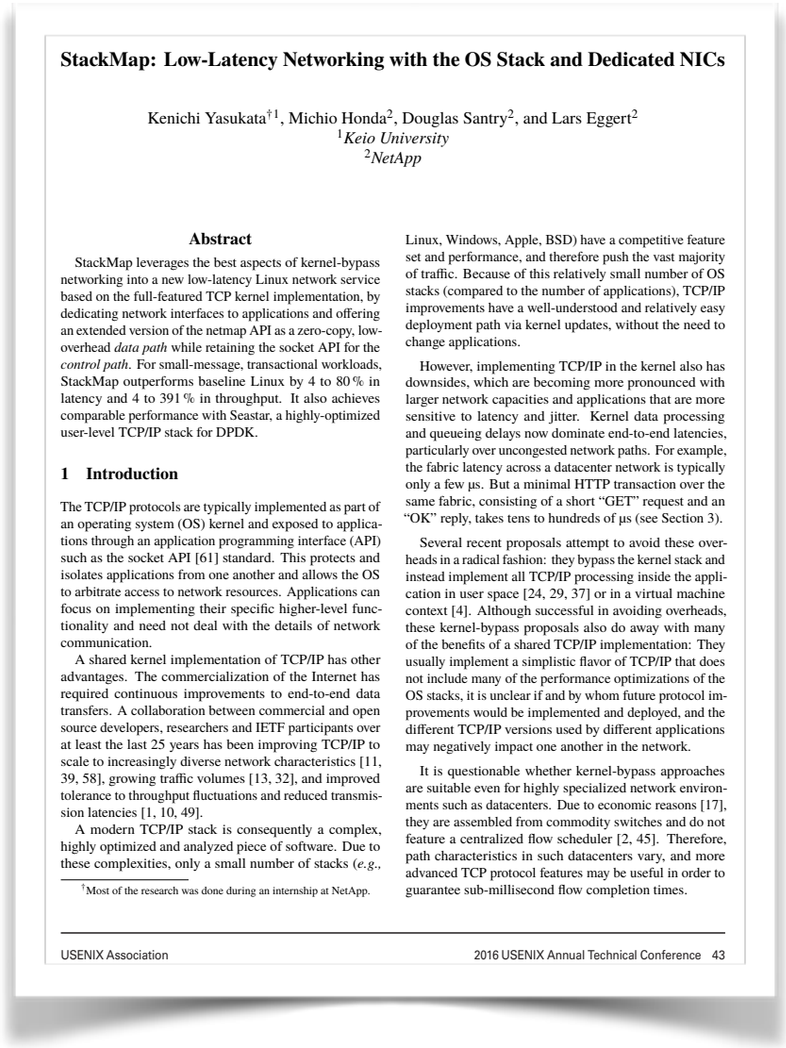


Source: L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012.

Colin Perkins | https://csperkins.org/ | Copyright © 2018

StackMap: Accelerating TCP with netmap

- Key insight: the TCP/IP stack processing is not expensive – inefficiencies are primarily system call overheads, copying data, and Socket API limitations with large number of file descriptors
- The netmap framework avoids the copies and reduces number of system calls, and gains in performance – but without the TCP/IP stack
- StackMap integrates the Linux kernel TCP/IP stack with netmap
  - Uses kernel TCP/IP stack for the control plane
  - Uses netmap for the data plane – new API
  - Like netmap, requires a dedicated network interface for each StackMap enabled application



StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata[†1], Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.*, Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

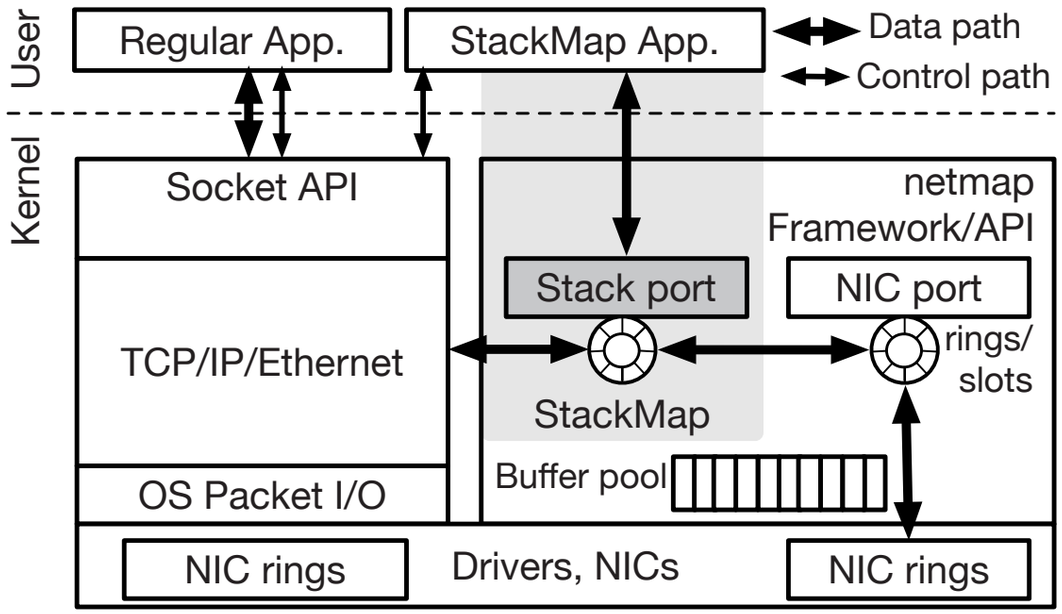
It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

[†]Most of the research was done during an internship at NetApp.

K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

Colin Perkins | https://csperkins.org/ | Copyright © 2018

StackMap Architecture



Source: K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016.

- Socket API for control: `socket()`, `bind()`, `listen()`, `accept()`, etc.
- Netmap API used for data
  - `STACKMAP_BUF()` updates netmap's circular buffer, and passes data through the TCP/IP stack for processing
- StackMap manages the buffer pool
  - Circular buffer used by netmap
  - If packet must be stored for retransmission, its buffer is swapped out of the netmap ring and replaced by another from the pool, until needed
    - Zero copy – just swaps pointers; buffer is already shared between netmap, kernel, and application
  - Also manages *scratchpad* data structure, to simplify iteration over data from on multiple connections

CC BY ND

StackMap API

```
1   struct sockaddr_in sin = { AF_INET, "10.0.0.1", INADDR_ANY };
2   int sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
3   bind(sd, &sin);
4   // prefix "stack" opens stack port for given interface
5   struct nm_desc *nmd = nm_open("stack:ix0");
6   connect(sd, dst_addr); /* non-blocking */
7   // transmit using ring 0 only, for this example
8   struct netmap_ring *ring = NETMAP_TXRING(nmd->nifp, 0);
9   uint32_t cur = ring->cur;
10  while (app_has_data && cur != ring->tail) {
11        struct netmap_slot *slot = &ring->slot[cur];
12        char *buf = STACKMAP_BUF(ring, slot->buf_index);
13        // place payload in buf, then
14        slot->fd = sd;
15        cur = nm_ring_next(ring, cur);
16  }
17  ring->head = ring->cur = cur;
18  ioctl(nmd->fd, NIOCTXSYNC);
```

Sockets API used to initiate connection

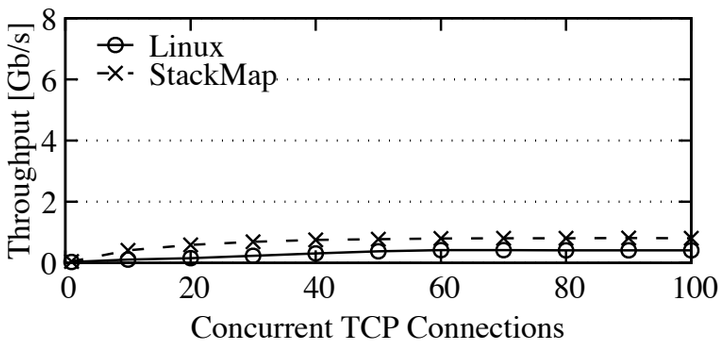
The netmap API is used to send and receive data

The `STACKMAP_BUF()` call is an extension that passes the data to the kernel TCP/IP stack, handles ACKs, retransmission, congestion control, etc.

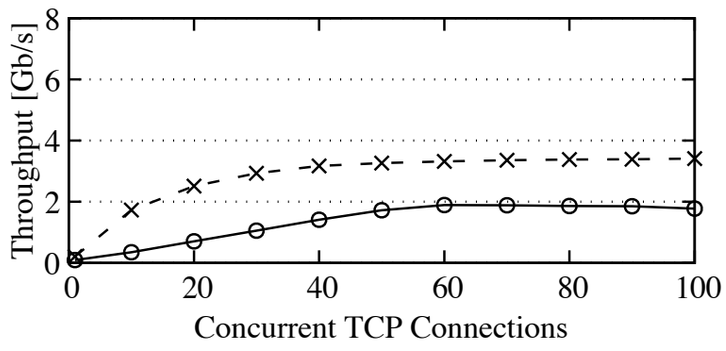
Source: K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016.

Colin Perkins | https://csperkins.org/ | Copyright © 2018

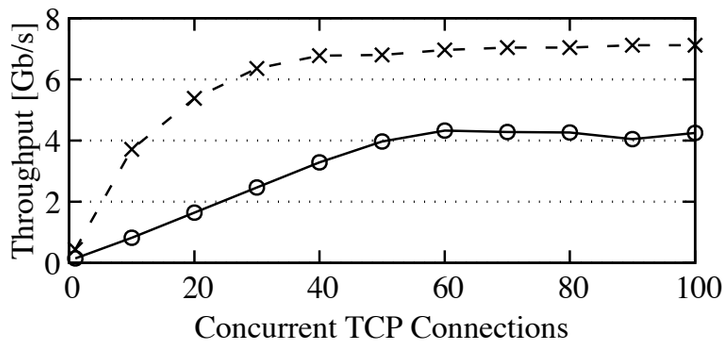
StackMap Performance



64 B Response Size



512 B Response Size



1280 B Response Size

- Linux with StackMap extensions outperforms standard Linux
- Primary benefits are avoiding copies and better scaling with concurrent flows: performance benefit increases with response size and number of connections

- Note: uses the full kernel TCP/IP stack – not a cut-down version – benefits due to new API, not reduced functionality

CC BY ND

Accelerating TCP: Other Approaches

- Numerous other attempts to accelerate TCP/IP stack processing exist:
  - Example: sandstorm builds on netmap; combines application, TCP/IP, and ethernet processing into a user-space library – builds highly optimised, special purpose, protocol stack for each application (see “Network Stack Specialization for Performance”)
  - Example: Google’s QUIC protocol aims to provide an alternative to TCP, with better performance, in a user space protocol running over UDP
  - …
- No clear consensus on the right approach
  - API changes to batch packet processing and reduce copies are clearly beneficial – how general purpose the resulting API needs to be is unclear
  - Pervasive encryption may reduce the benefits – you need to make a copy while decrypting, and many of these approaches benefit from zero-copy stacks



Network Stack Specialization for Performance

Ilias Marinos
University of Cambridge
Ilias.Marinos@cl.cam.ac.uk

Robert N.M. Watson
University of Cambridge
Robert.Watson@cl.cam.ac.uk

Mark Handley
University College London
M.Handley@cs.ucl.ac.uk

Abstract

Contemporary network stacks are masterpieces of generality, supporting many edge-node and middle-node functions. Generality comes at a high performance cost: current APIs, memory models, and implementations drastically limit the effectiveness of increasingly powerful hardware. Generality has historically been required so that individual systems could perform many functions. However, as providers have scaled services to support millions of users, they have transitioned toward thousands (or millions) of dedicated servers, each performing a few functions. We argue that the overhead of generality is now a key obstacle to effective scaling, making specialization not only viable, but necessary.

We present Sandstorm and Namestorm, web and DNS servers that utilize a clean-slate userspace network stack that exploits knowledge of application-specific workloads. Based on the netmap framework, our novel approach merges application and network-stack memory models, aggressively amortizes protocol-layer costs based on application-layer knowledge, couples tightly with the NIC event model, and exploits microarchitectural features. Simultaneously, the servers retain use of conventional programming frameworks. We compare our approach with the FreeBSD and Linux stacks using the nginx web server and NSD name server, demonstrating 2–10× and 9× improvements in web-server and DNS throughput, lower CPU usage, linear multicore scaling, and saturated NIC hardware.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design — *Network communications*

General Terms: Design, performance

Keywords: Network stacks; network performance; network-stack specialization; clean-slate design

1. INTRODUCTION

Conventional network stacks were designed in an era where individual systems had to perform multiple diverse functions. In the last decade, the advent of cloud computing and the ubiquity of networking has changed this model; today, large content providers serve hundreds of millions of customers. To scale their systems, they are forced to employ many thousands of servers, with each providing only a single network service. Yet most content is still served with conventional general-purpose network stacks.

These general-purpose stacks have not stood still, but today’s stacks are the result of numerous incremental updates on top of codebases that were originally developed in the early 1990s. Arguably, these network stacks have proved to be quite efficient, flexible, and reliable, and this is the reason that they still form the core of contemporary networked systems. They also provide a stable programming API, simplifying software development. But this generality comes with significant costs, and we argue that the overhead of generality is now a key obstacle to effective scaling, making specialization not only viable, but necessary.

In this paper we revisit the idea of specialized network stacks. In particular, we develop Sandstorm, a specialized userspace stack for serving static web content, and Namestorm, a specialized stack implementing a high performance DNS server. More importantly, however, our approach does not simply shift the network stack to userspace: we also promote tight integration and specialization of application and stack functionality, achieving cross-layer optimizations antithetical to current design practices.

Servers such as Sandstorm could be used for serving images such as the Facebook logo, as OCSP [20] responders for certificate revocations, or as front end caches to popular dynamic content. This is a role that conventional stacks should be good at: nginx [6] uses the `sendfile()` system call to hand over serving static content to the operating system. FreeBSD and Linux then implement zero-copy stacks, at least for the payload data itself, using scatter-gather to directly DMA the payload from the disk buffer cache to the NIC. They also utilize the features of smart network hardware, such as TCP Segmentation Offload (TSO) and Large Receive Offload (LRO) to further improve performance. With such optimizations, nginx does perform well, but as we will demonstrate, a specialized stack can outperform it by a large margin.

Namestorm is aimed at handling extreme DNS loads, such as might be seen at the root nameservers, or when a server is under a high-rate DDoS attack. The open-source state of the art here is NSD [5], which combined with a modern OS that minimizes data copies when sending and receiving UDP packets, performs well. Namestorm, however, can outperform it by a factor of nine.

Our userspace web server and DNS server are built upon FreeBSD’s netmap [31] framework, which directly maps the NIC buffer rings to userspace. We will show that not only is it possible for a specialized stack to beat nginx, but on data-center-style networks when serving small files typical of many web pages, it can achieve

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM’14, August 17–22, 2014, Chicago, Illinois, USA.
Copyright 2014 ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626311.

I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. Proceedings of the SIGCOMM Conference, Chicago, IL, USA, August 2014. ACM. DOI: 10.1145/2619239.2626311 (Open access via http://dl.acm.org/authorize?N71202)

CC BY ND

Conclusions

- Improvements to network performance relative to CPU performance highlight limitations of Sockets API for high performance – ongoing work to find good new APIs
- Beware: it's very hard to beat TCP – incredibly well tuned, secure, and well maintained
  - TCP highly optimised over many years – handles edge cases in security and congestion control that are non-obvious
    - IETF RFC 7414 ("A Roadmap for TCP Specification Documents") is 55 pages, and references 150 other documents – textbook explanations of TCP omit much important detail
  - Likely better to optimise kernel TCP stack and API, than end up stuck on a poorly maintained user-space stack

CC BY ND

Further Reading – For Tutorial



netmap: a novel framework for fast packet I/O

Luigi Rizzo, *Università di Pisa, Italy*[*]
Proceedings of the 2012 USENIX Annual Technical Conference, June 2012.
https://www.usenix.org/conference/atc12/ [†]

Abstract

Many applications (routers, traffic monitors, firewalls, etc.) need to send and receive packets at line rate even on very fast links. In this paper we present *netmap*, a novel framework that enables commodity operating systems to handle the millions of packets per seconds traversing 1..10 Gbit/s links, without requiring custom hardware or changes to applications.

In building *netmap*, we identified and successfully reduced or removed three main packet processing costs: per-packet dynamic memory allocations, removed by preallocating resources; system call overheads, amortized over large batches; and memory copies, eliminated by sharing buffers and metadata between kernel and userspace, while still protecting access to device registers and other kernel memory areas. Separately, some of these techniques have been used in the past. The novelty in our proposal is not only that we exceed the performance of most of previous work, but also that we provide an architecture that is tightly integrated with existing operating system primitives, not tied to specific hardware, and easy to use and maintain.

netmap has been implemented in FreeBSD and Linux for several 1 and 10 Gbit/s network adapters. In our prototype, a single core running at 900 MHz can send or receive 14.88 Mpps (the peak packet rate on 10 Gbit/s links). This is more than 20 times faster than conventional APIs. Large speedups (5x and more) are also achieved on user-space Click and other packet forwarding applications using a libpcap emulation library running on top of *netmap*.

[*]This work was funded by the EU FP7 project CHANGE (257422).
[†]USENIX plans to publish this paper on the *Proceedings of the 2012 USENIX Annual Technical Conference,* which will be available at this URL after June 13, 2012. You may also find this paper, with related material, on the author's site, http://info.iet.unipi.it/ luigi/netmap/

1 Introduction

General purpose OSes provide a rich and flexible environment for running, among others, many packet processing and network monitoring and testing tasks. The high rate raw packet I/O required by these applications is not the intended target of general purpose OSes. Raw sockets, the Berkeley Packet Filter [14] (BPF), the AF_SOCKET family, and equivalent APIs have been used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is inadequate for the millions of packets per second (*pps*) that can be present on 1..10 Gbit/s links. In search of better performance, some systems (see Section 3) either run completely in the kernel, or bypass the device driver and the entire network stack by exposing the NIC's data structures to user space applications. Efficient as they may be, many of these approaches depend on specific hardware features, give unprotected access to hardware, or are poorly integrated with the existing OS primitives.

The *netmap* framework presented in this paper combines and extends some of the ideas presented in the past trying to address their shortcomings. Besides giving huge speed improvements, *netmap* does not depend on specific hardware[1], has been fully integrated in FreeBSD and Linux with minimal modifications, and supports unmodified libpcap clients through a compatibility library.

One metric to evaluate our framework is performance: in our implementation, moving one packet between the wire and the userspace application has an amortized cost of less than 70 CPU clock cycles, which is at least one order of magnitude faster than standard APIs. In other words, a single core running at 900 MHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link. The same core running at 150 MHz is well above the capacity

[1]*netmap* can give isolation even without hardware mechanisms such as IOMMU or VMDq, and is orthogonal to hardware offloading and virtualization mechanisms (checksum, TSO, LRO, VMDc, etc.)

1

L. Rizzo. netmap: a novel framework for fast packet I/O. In Proceedings of the USENIX Annual Technical Conference, Boston, MA, USA, June 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo (paper and video of presentation)



StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata[†1], Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.,* Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

[†]Most of the research was done during an internship at NetApp.

K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In Proceedings of the USENIX Annual Technical Conference, Denver, CO, USA, June 2016. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata

Colin Perkins | https://csperkins.org/ | Copyright © 2018