Kernel Bypass

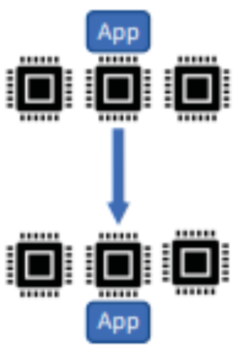
ECE/CS598HPN

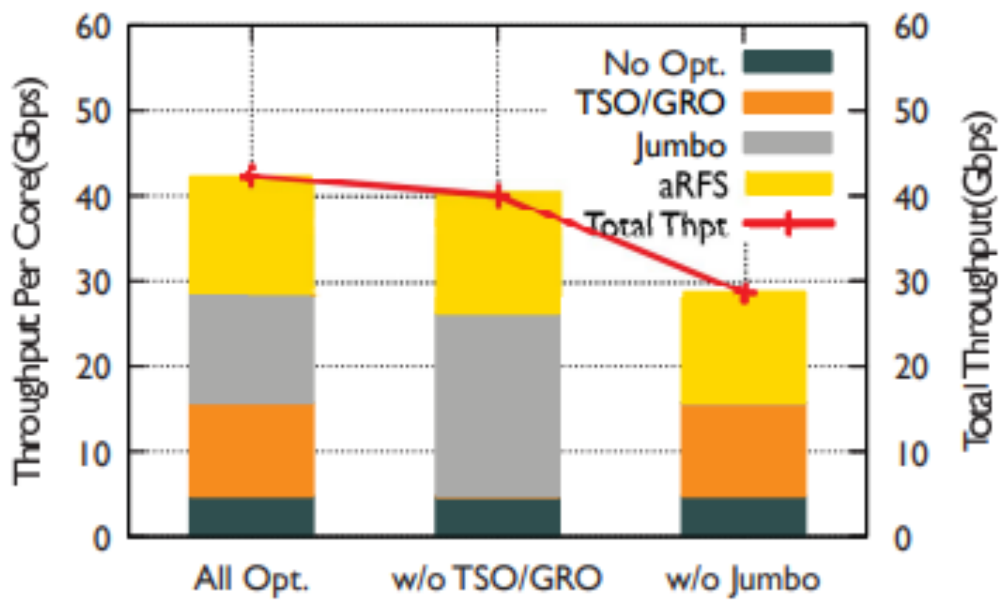
Radhika Mittal

Performance overhead in kernel stack

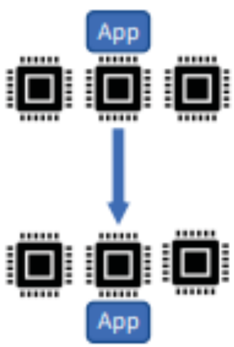
- Protocol processing
- Data copy
- Cache contention (between flows sharing same NUMA node)
- CPU scheduling overheads (locking, context switching)
- Interrupts
- Managing heavy datastructures (skbs)

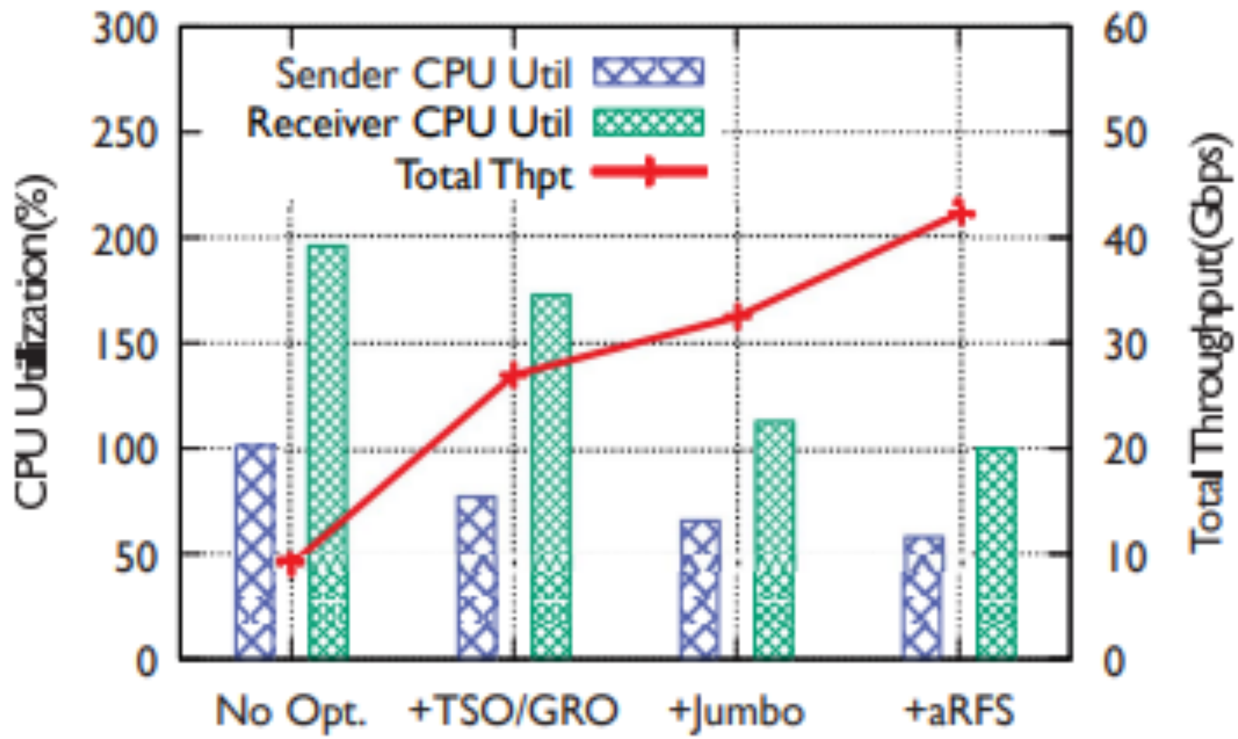
Results from "understanding host network stack"





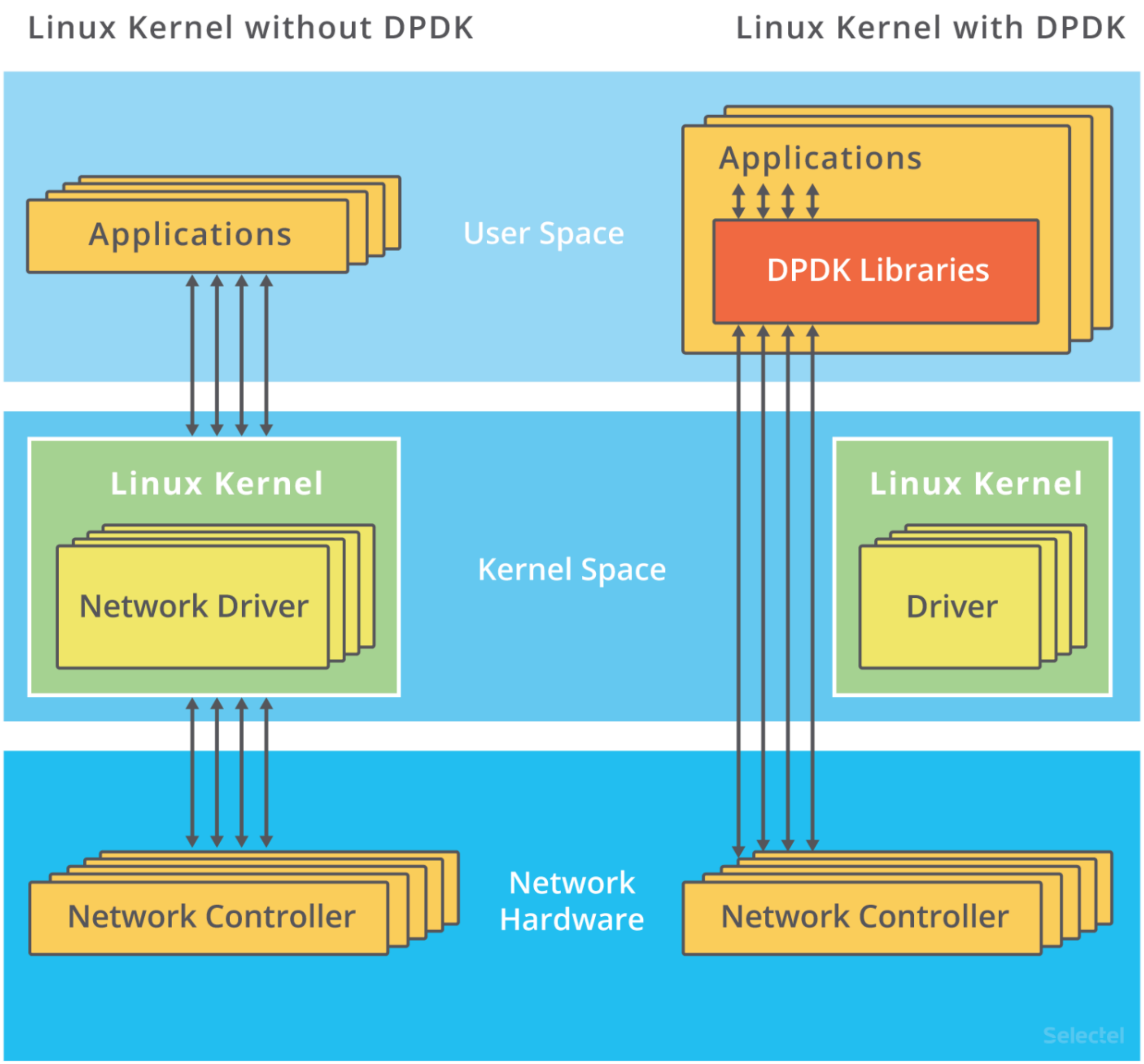
Results from "understanding host network stack"





Kernel Bypass Packet I/O

Dataplane Development Kit (DPDK)



Source: https://blog.selectel.com/introduction-dpdk-architecture-principles/

Dataplane Development Kit (DPDK)

- User-space packet processing (kernel bypass).
  - Avoid context switching overhead.
- Poll Mode Driver (PMD).
  - Avoid interrupt processing overhead.
  - *Keeps a core busy.*
- Memory usage optimizations
  - Light-weight *mbufs*.
  - Memory pools that use hugepages, cache alignment, etc.
  - Lockless ring buffers.

Other examples

- NetMap
  - In-kernel module for efficient packet processing.
  - Light-weight packet buffers.
  - Fewer memory copies.

- Packet Shader
  - Modified packet I/O engine in the kernel.
  - Fetches packets through a combination of interrupts and polling.
  - Processes packets using GPU in userspace.

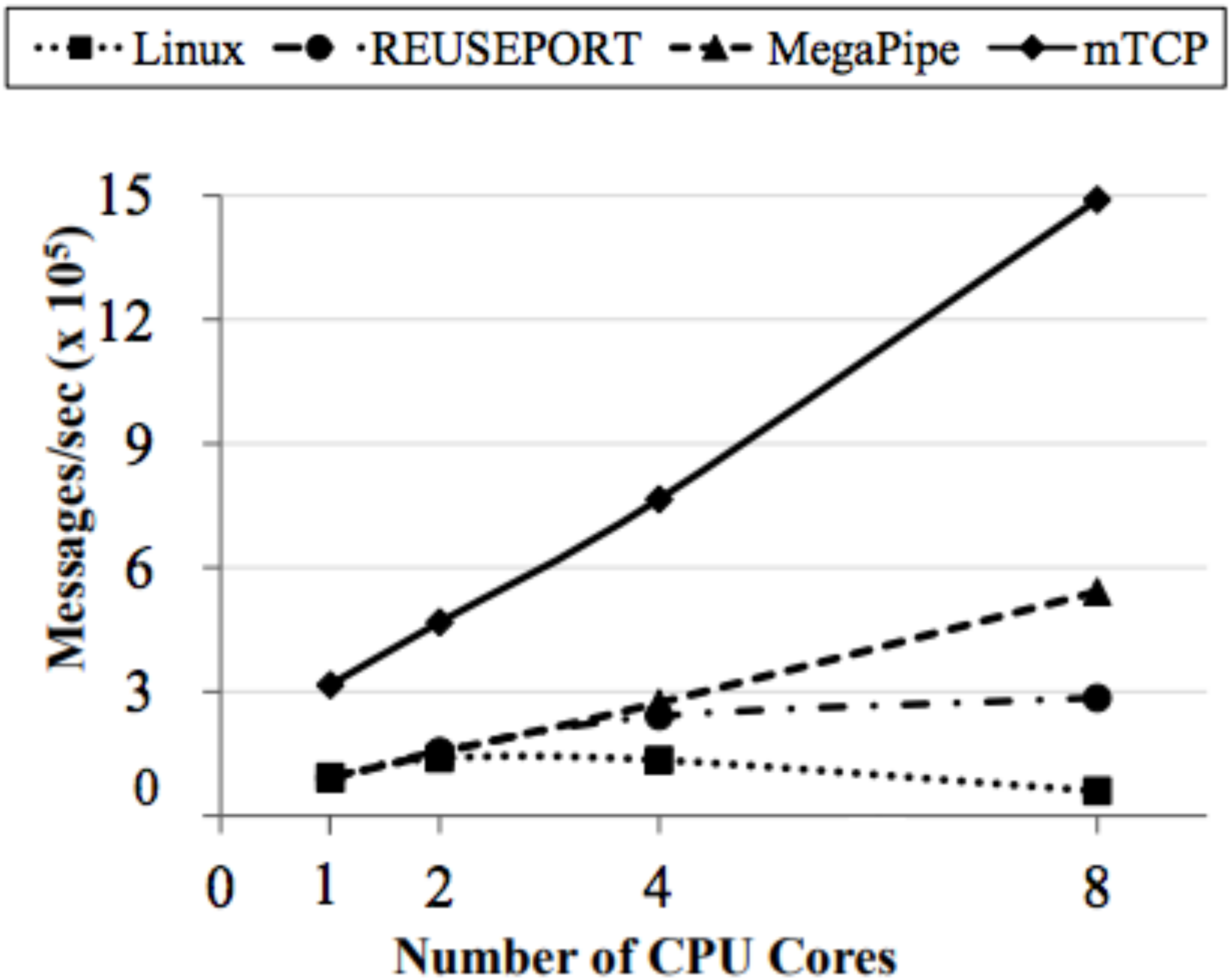
Kernel Bypass Packet I/O Engine

- Provide mechanisms for delivering packets to user space.
- Do not implement a network stack.

mTCP

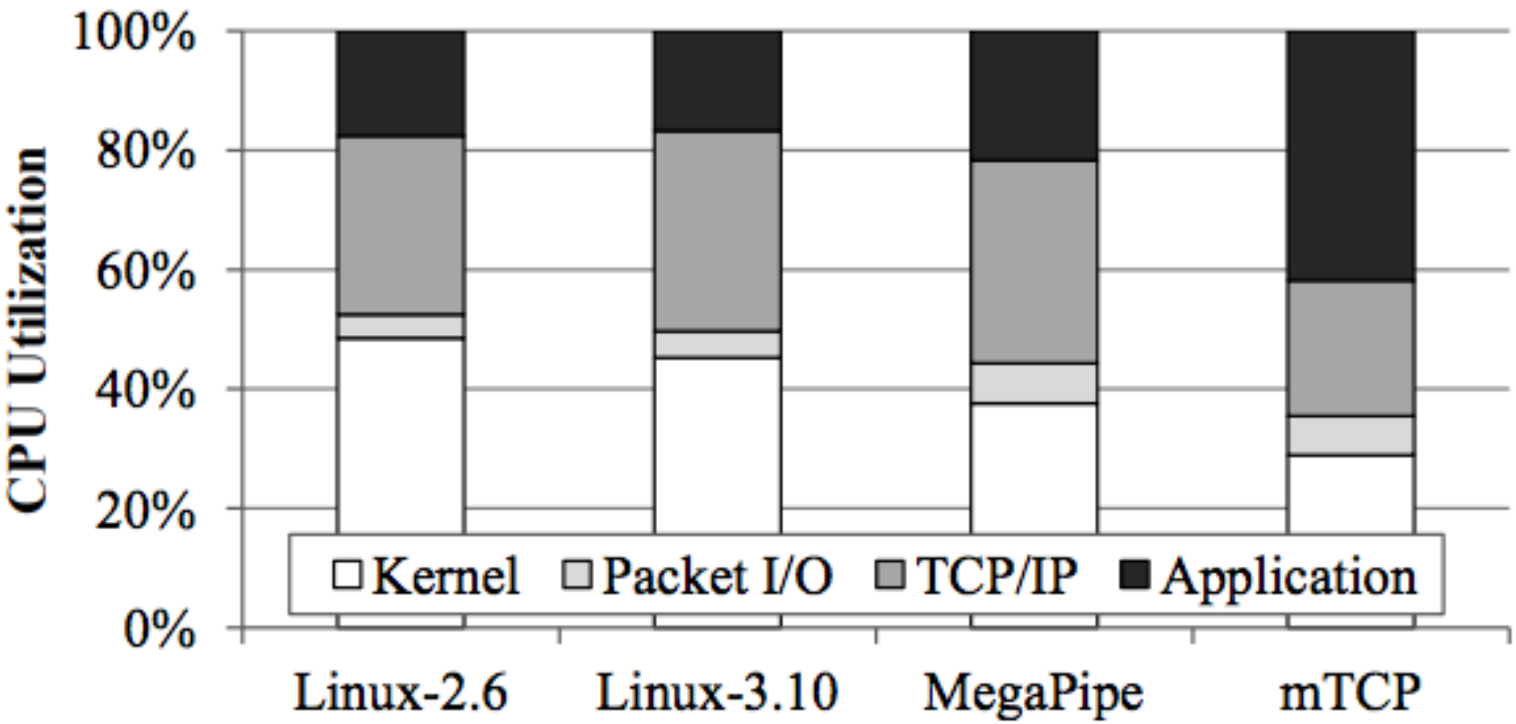
- User-space TCP/IP stack built over kernel-bypass packet I/O engines.
  - Implementation in paper over PacketShader.
  - DPDK based implementation also available.

mTCP



Linux
REUSEPORT
MegaPipe
mTCP
Messages/sec (x 10⁵)
15
12
9
6
3
0
0
1
2
4
8
Number of CPU Cores

mTCP



100%
80%
60%
40%
20%
0%
CPU Utilization
Kernel
Packet I/O
TCP/IP
Application
Linux-2.6
Linux-3.10
MegaPipe
mTCP

mTCP -- Issues

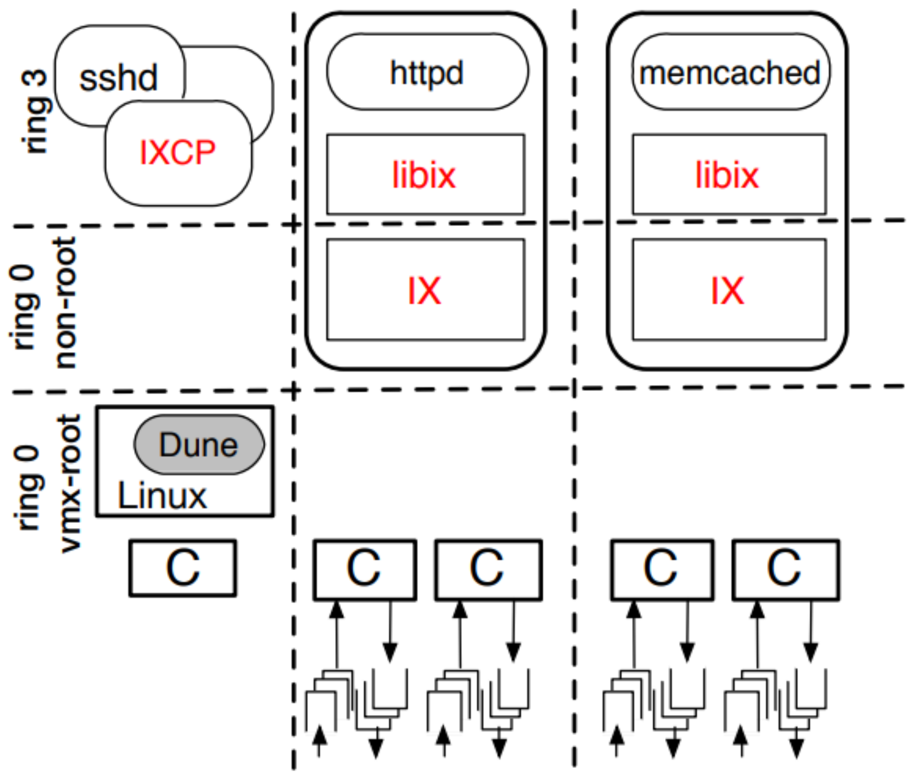
- Dedicated threads for the TCP stack.
  - Avoid intrusive inter-twining of application and TCP processing.
  - Batching to reduce switching overheads.
  - Adds latency.
- Security vulnerabilities with user-space network stack.

IX (OSDI'14)

- *Protected kernel-bypass.*

- Separation of control plane and dataplane
  - Control plane handles resource allocation (cores, memory, network queues).
  - Three-way isolation: IX control plane, dataplane (guest), and untrusted user code.
  - Hardware virtualization techniques to expose resources to dataplane.

IX (OSDI'14)

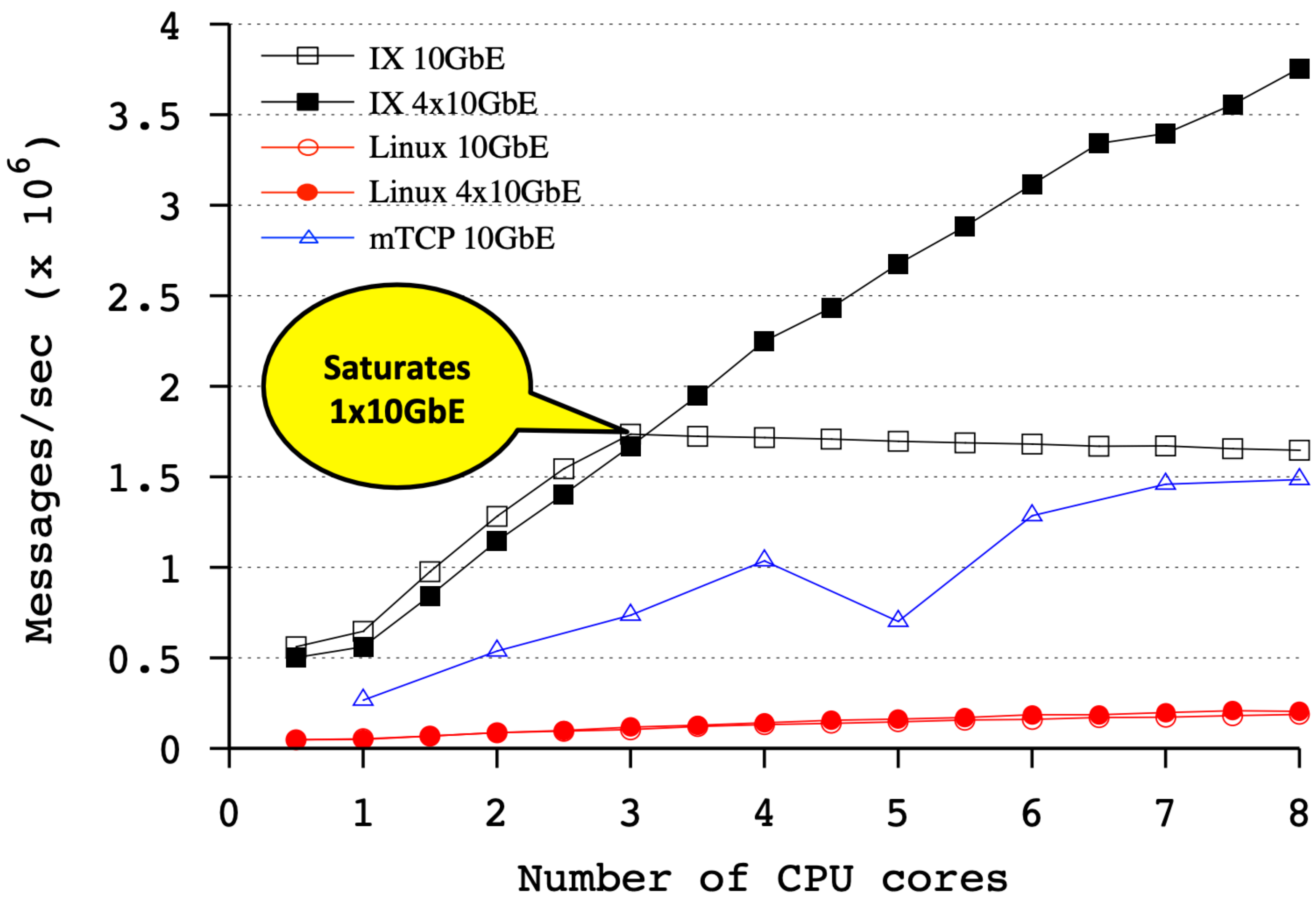


(a) Protection and separation of control and data plane.

IX (OSDI'14)

- Run to completion packet processing in dataplane.
  - Adaptive Batching
  - Zero-copy
  - Synchronization-free processing
  - Implemented over DPDK

IX Performance



IX Limitations

- [TAS, EuroSys'19] Might still have additional protocol processing overheads….
  - TCP packet processing in one monolithic block.
    - Large amount of per-connection state, many branches, increased cache footprint.

- Non-socket API

- Enforcing zero-copy requires application and dataplane kernel to coordinate on buffer management.
  - Application must not mutate content of a packet until it's acknowledged.
  - (Similar issues with kernel zero-copy mechanisms).

TAS: TCP Acceleration as an OS service (EuroSys'19)

Slides from TAS authors.

RPCs are Essential in the Datacenter

Remote procedure calls (RPCs) are a common building block for datacenter applications

Scenario: An efficient key-value store in a datacenter

1. Low tail latency is crucial
2. Thousands of connections per machine
3. Both the application writer and datacenter operator want the full feature set of TCP
   a) Developers want the convenience of *sockets* and *in-order delivery*
   b) Operators want *flexibility* and strong *policy enforcement*

You might want to simply go with Linux…

Linux provides the features we want

sockets *in-order delivery* *flexibility* *policy enforcement*

But at what cost?

You might want to simply go with Linux…

Linux provides the features we want

sockets *in-order delivery* *flexibility* *policy enforcement*

But at what cost?

A simple KVS model:

256B RPC request/response over Linux TCP
250 application cycles per RPC

You might want to simply go with Linux…

Linux provides the features we want

sockets *in-order delivery* *flexibility* *policy enforcement*

But at what cost?

A simple KVS model:

256B RPC request/response over Linux TCP

250 application cycles per RPC

8,300 Total CPU Cycles per RPC

We're only doing a small amount of useful computation!

Why is Linux slow?

Application and kernel co-location → *System call and cache pollution overheads*

Executes entire TCP state machine → *Complicated data path*

State in multiple cache lines → *Poor cache efficiency, unscalable*

Why not kernel-bypass?

NIC interface is optimized, bottlenecks are in OS

Arrakis (OSDI ‘14), *mTCP* (NSDI ‘14), *Stackmap* (ATC ‘16)

Do network processing in userspace

- Expose the NIC interface to the application
- Hardware I/O virtualization

Why not kernel-bypass?

NIC interface is optimized, bottlenecks are in OS

Arrakis (OSDI '14), *mTCP* (NSDI '14), *Stackmap* (ATC '16)

Do network processing in userspace

- Expose the NIC interface to the application
- Hardware I/O virtualization

☑ Avoid OS overheads, can specialize stack

Why not kernel-bypass?

NIC interface is optimized, bottlenecks are in OS

Arrakis (OSDI '14), *mTCP* (NSDI '14), *Stackmap* (ATC '16)

Do network processing in userspace

- Expose the NIC interface to the application
- Hardware I/O virtualization

☑ Avoid OS overheads, can specialize stack

☒ Operators have to trust application code

☒ Little flexibility for operators to change or update network stack

Why not RDMA? (next week)

Remote Direct Memory Access:

- Interface: **one-sided** and **two-sided** operations in NIC hardware
- RPCs and sockets implemented on top of basic RDMA primitives

Why not RDMA? (next week)

Remote Direct Memory Access:

Interface: **one-sided** and **two-sided** operations in NIC hardware

RPCs and sockets implemented on top of basic RDMA primitives

☑ Minimize or bypass CPU overhead

Why not RDMA? (next week)

Remote Direct Memory Access:

Interface: **one-sided** and **two-sided** operations in NIC hardware

RPCs and sockets implemented on top of basic RDMA primitives

- ☑ Minimize or bypass CPU overhead
- ☒ Lose software procotol flexibility
- ☒ Bad fit for many-to-many RPCs
- ☒ RDMA congestion control (DCQCN) doesn‘t work well at scale

TAS: TCP Acceleration as an OS Service

An open source, drop-in, highly efficient RPC acceleration service

No additional NIC hardware required

Compatible with all applications that already use sockets

Operates as a userspace OS service using dedicated cores for packet processing

Leverages the benefits and flexibility of kernel bypass with better protection

TAS accelerates TCP processing for RPCs while providing all the desired features

Sockets *In-order delivery* *Flexibility* *Policy enforcement*

Why is Linux slow?

Application and kernel co-location → *System call and cache pollution overheads*

Executes entire TCP state machine → *Complicated data path*

State in multiple cache lines → *Poor cache efficiency, unscalable*

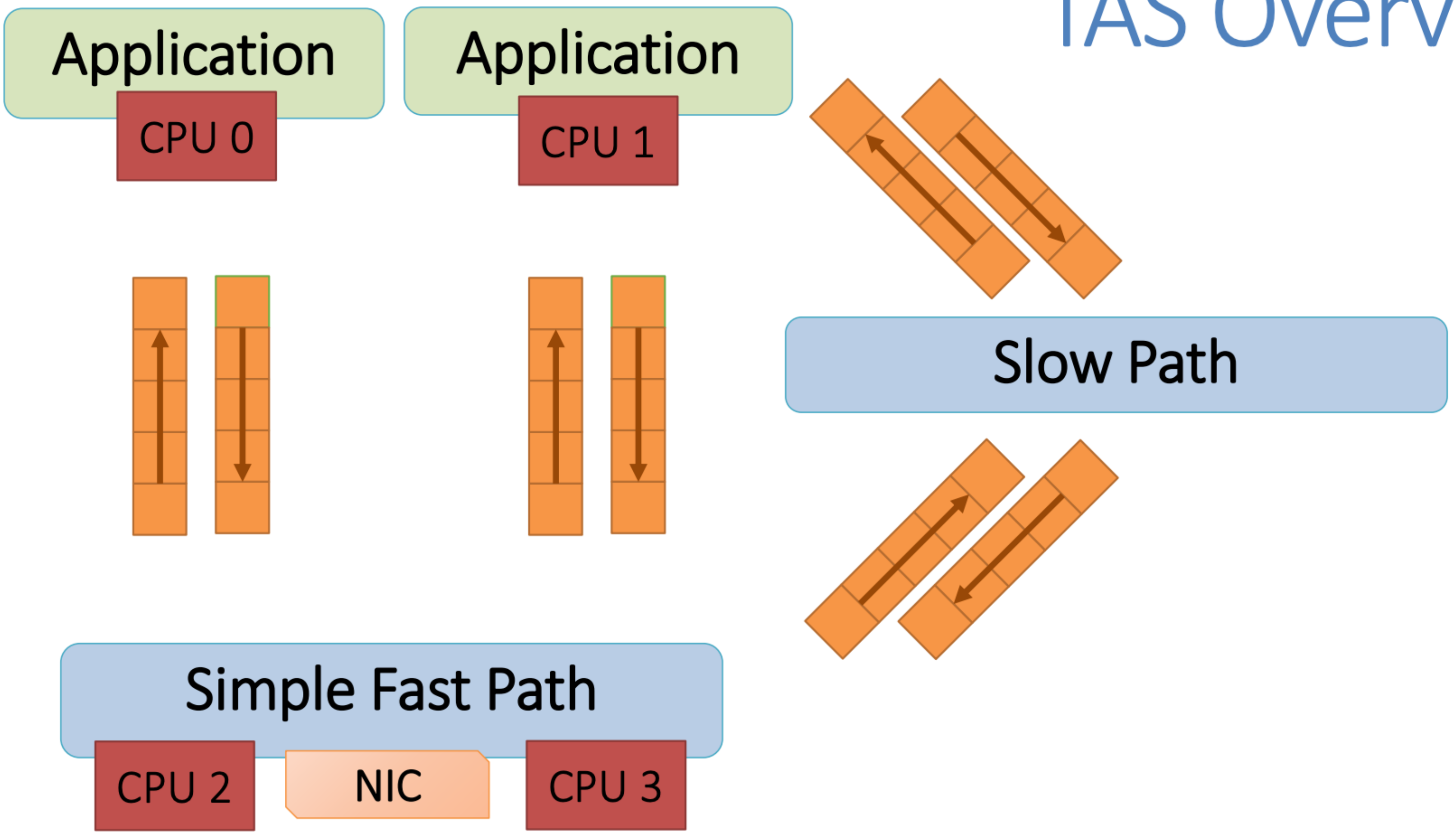
How does TAS fix it?

System call and cache pollution overheads → Dedicate cores for network stack

Complicated data path → Separate simple fast path and slow path

Poor cache efficiency, unscalable → Minimize and localize connection state

TAS Overview



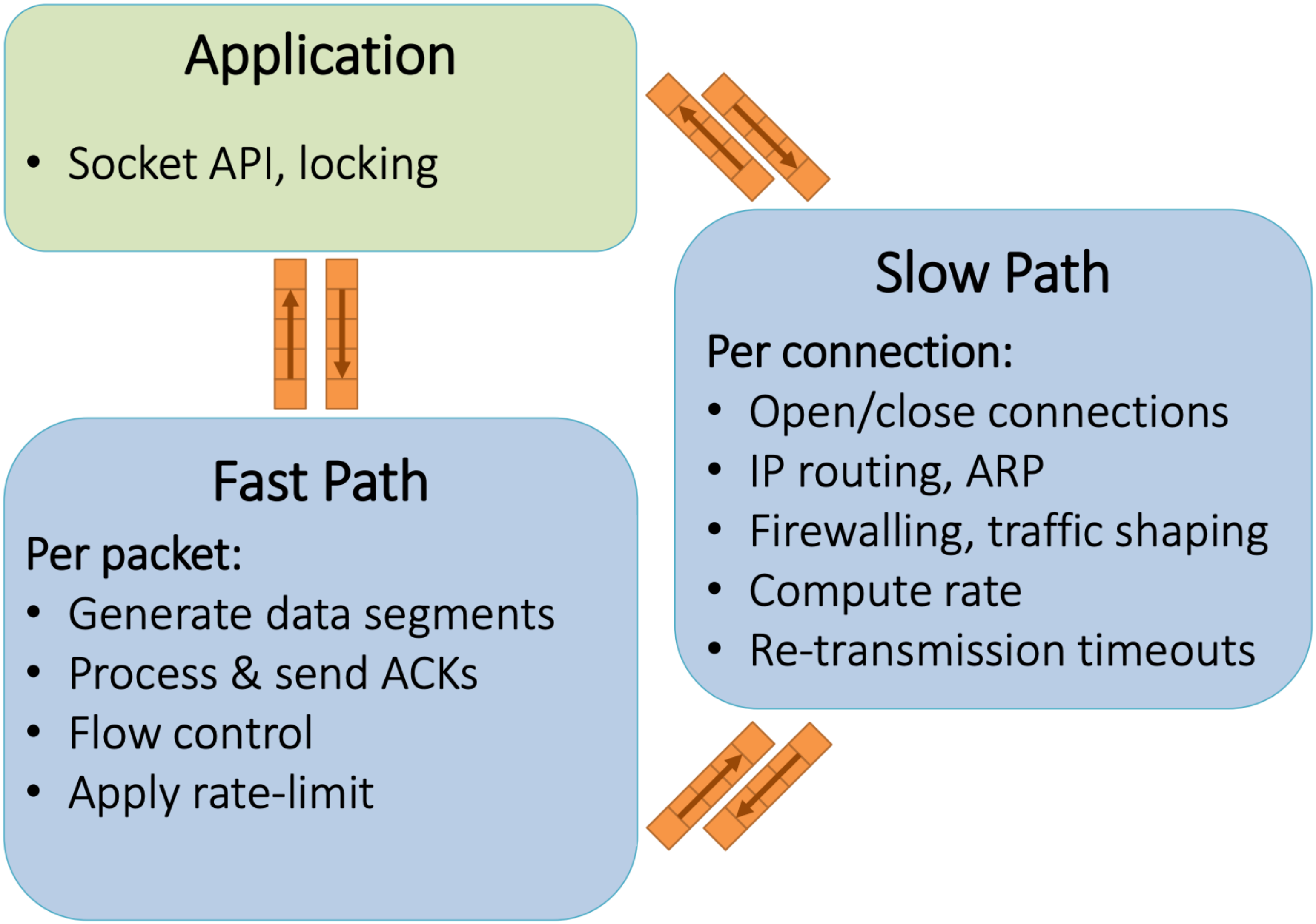
Dividing Functionality

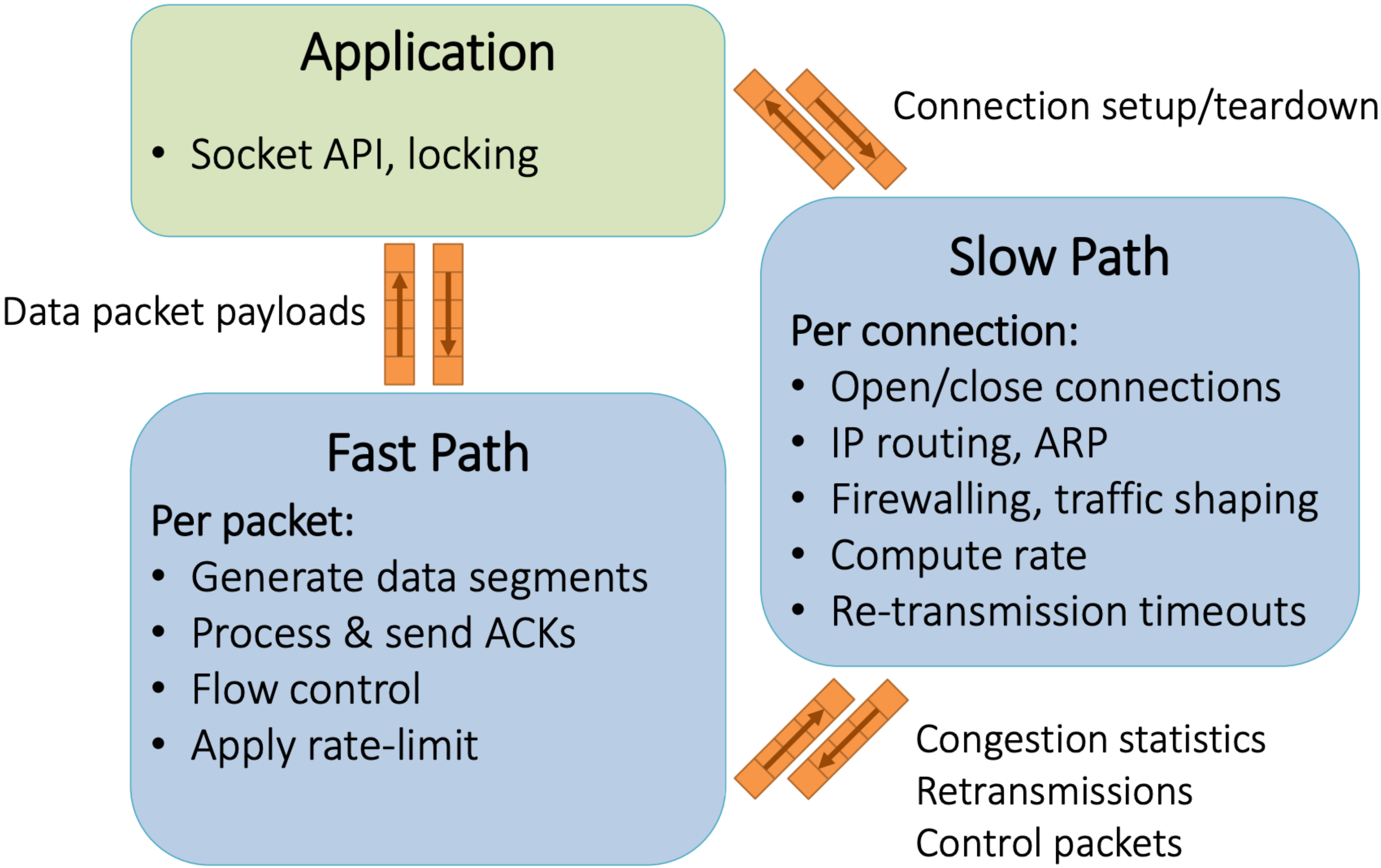
Linux Kernel TCP Stack

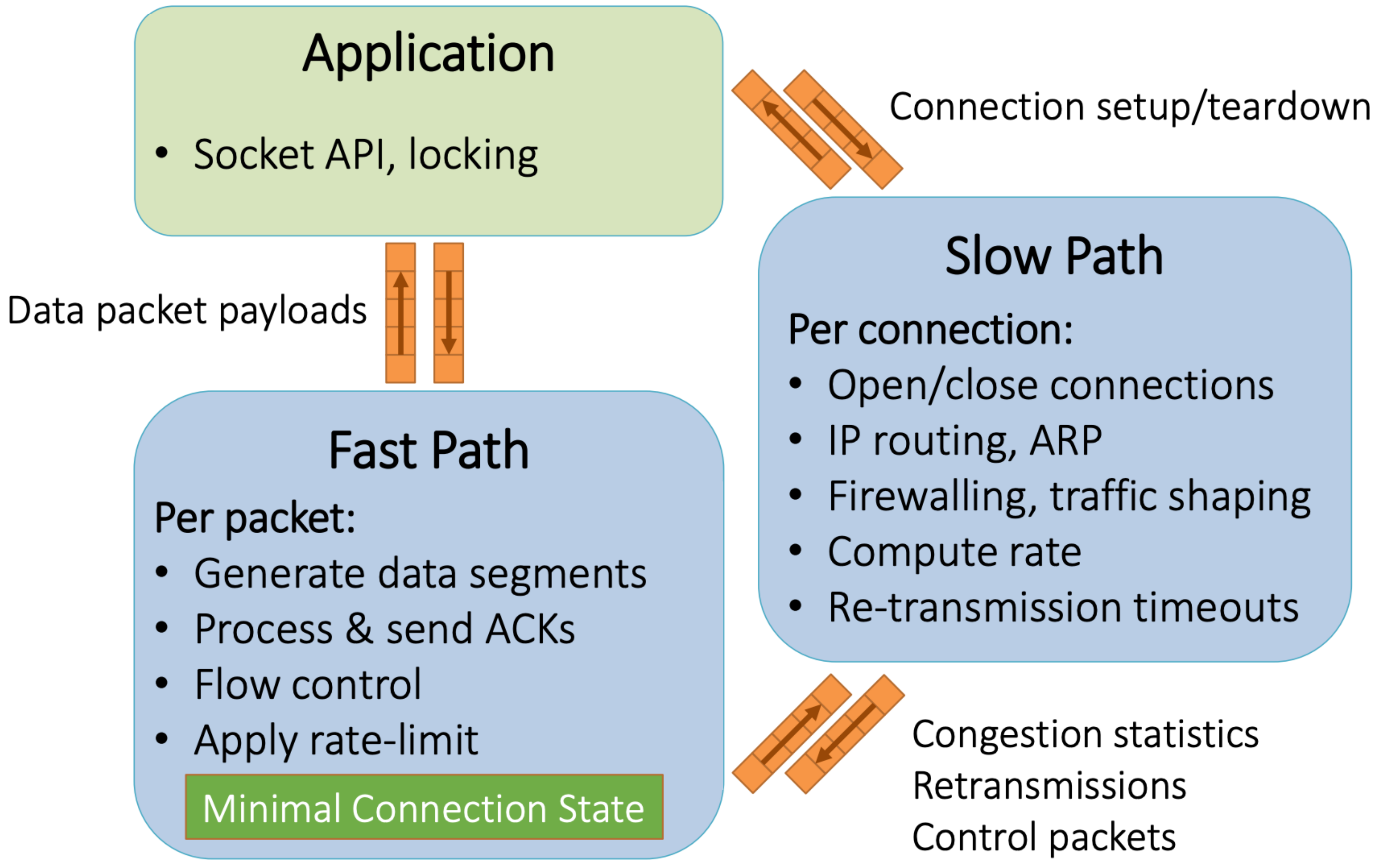
- Open/close connections

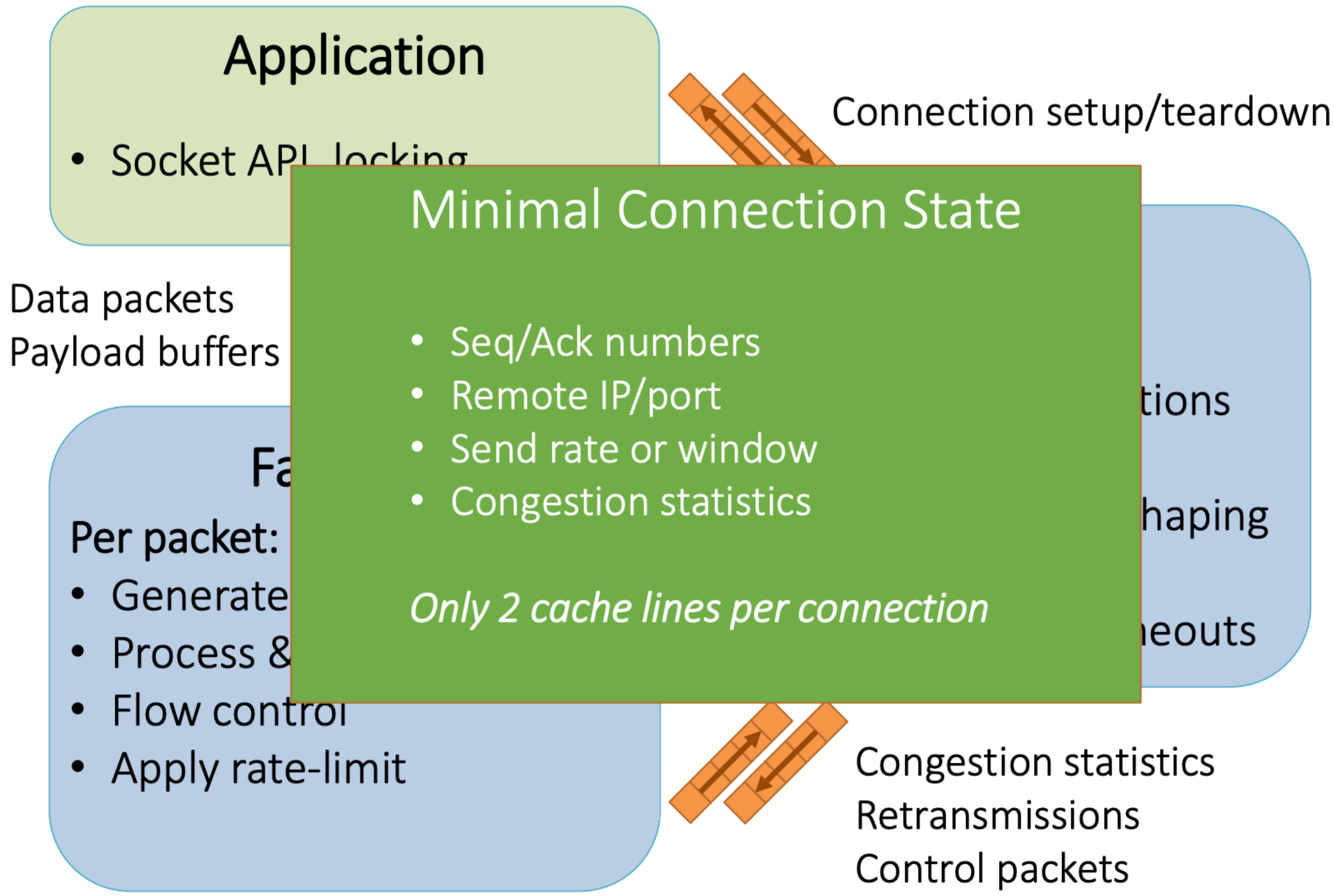
Per packet:

- Socket API, locking
- IP routing, ARP
- Firewalling, traffic shaping
- Generate data segments
- Congestion control
- Flow control
- Process & send ACKs
- Re-transmission timeouts


Application
• Socket API, locking
Slow Path
Per connection:
• Open/close connections
• IP routing, ARP
• Firewalling, traffic shaping
• Compute rate
• Re-transmission timeouts
Fast Path
Per packet:
• Generate data segments
• Process & send ACKs
• Flow control
• Apply rate-limit

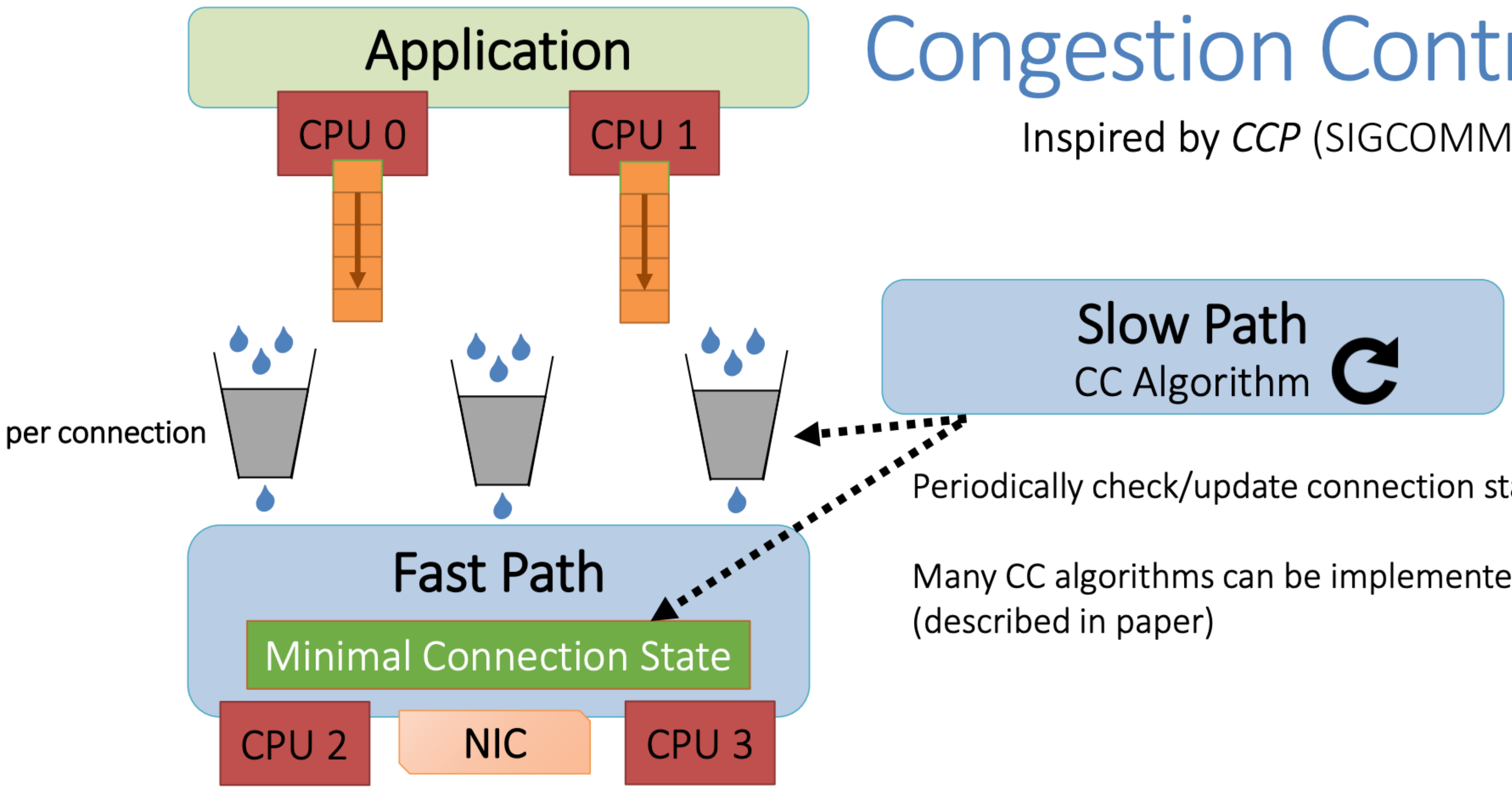

Application
• Socket API, locking
Connection setup/teardown
Data packet payloads
Slow Path
Per connection:
• Open/close connections
• IP routing, ARP
• Firewalling, traffic shaping
• Compute rate
• Re-transmission timeouts
Fast Path
Per packet:
• Generate data segments
• Process & send ACKs
• Flow control
• Apply rate-limit
Congestion statistics
Retransmissions
Control packets


Application
• Socket API, locking
Connection setup/teardown
Data packet payloads
Slow Path
Per connection:
• Open/close connections
• IP routing, ARP
• Firewalling, traffic shaping
• Compute rate
• Re-transmission timeouts
Fast Path
Per packet:
• Generate data segments
• Process & send ACKs
• Flow control
• Apply rate-limit
Minimal Connection State
Congestion statistics
Retransmissions
Control packets


Application
• Socket API locking
Connection setup/teardown
Minimal Connection State
• Seq/Ack numbers
• Remote IP/port
• Send rate or window
• Congestion statistics
Only 2 cache lines per connection
Data packets
Payload buffers
tions
haping
eouts
Fa
Per packet:
• Generate
• Process &
• Flow control
• Apply rate-limit
Congestion statistics
Retransmissions
Control packets

Congestion Control

Inspired by *CCP* (SIGCOMM '18)



Periodically check/update connection state

Many CC algorithms can be implemented (described in paper)

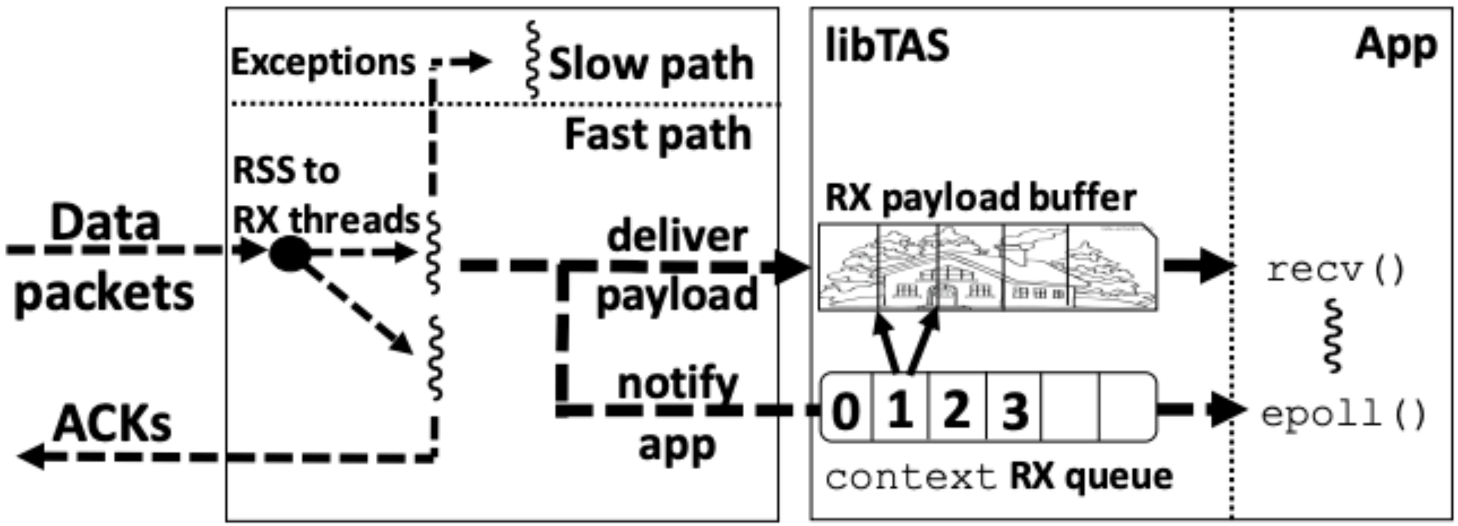


Figure 1. TAS receive flow.

- Application runs on separate core.
- Not zero-copy.

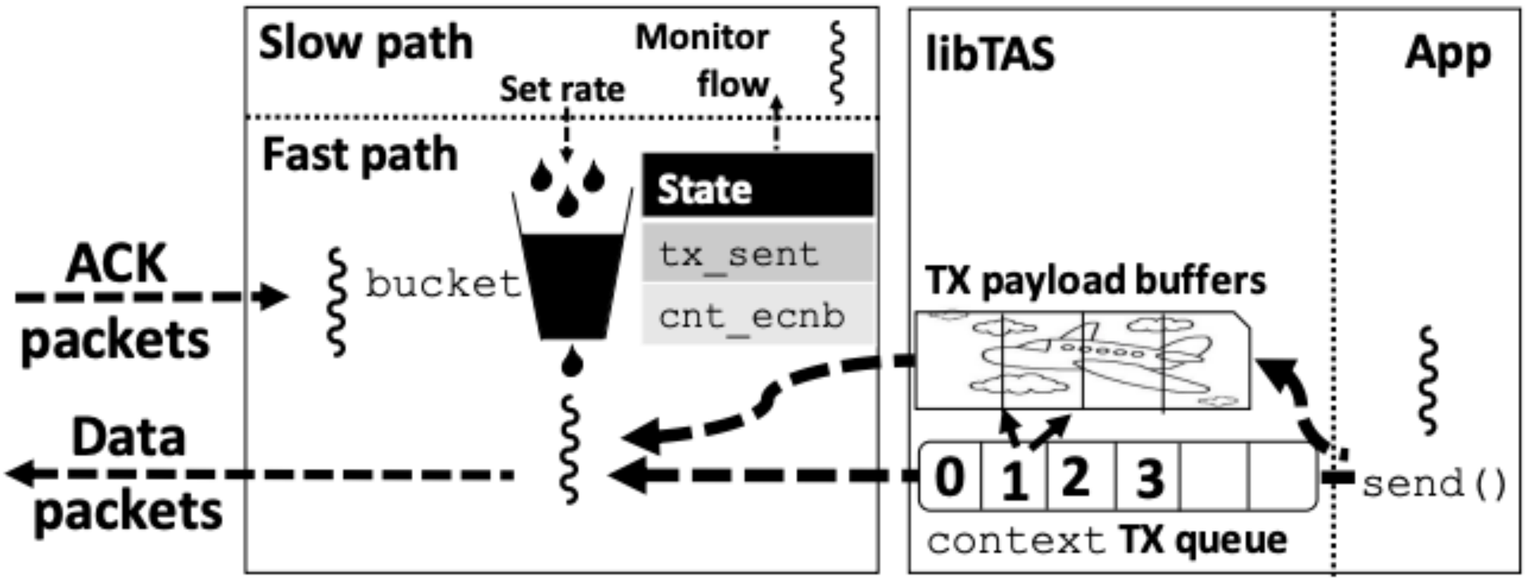
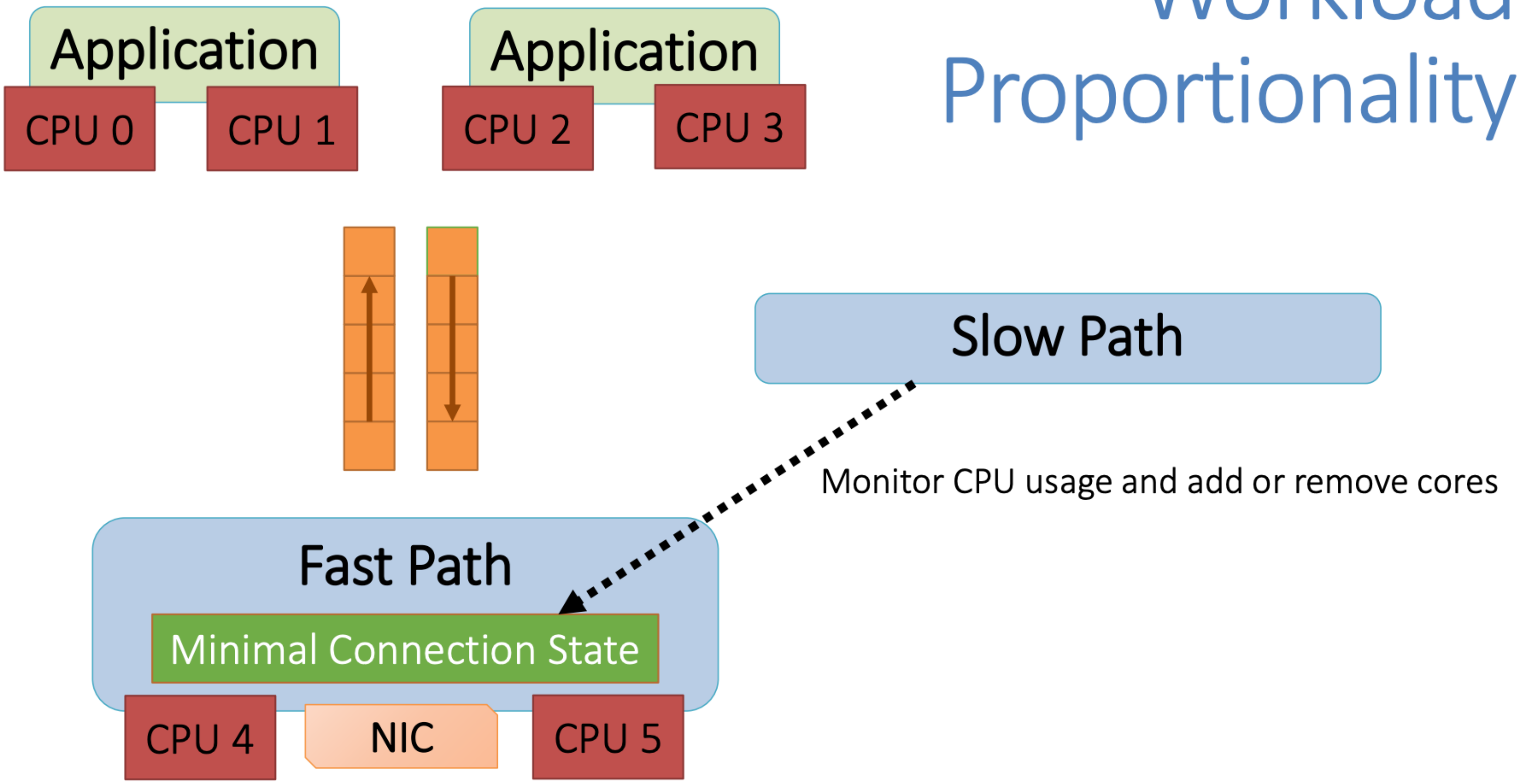


Figure 2. TAS transmit flow.

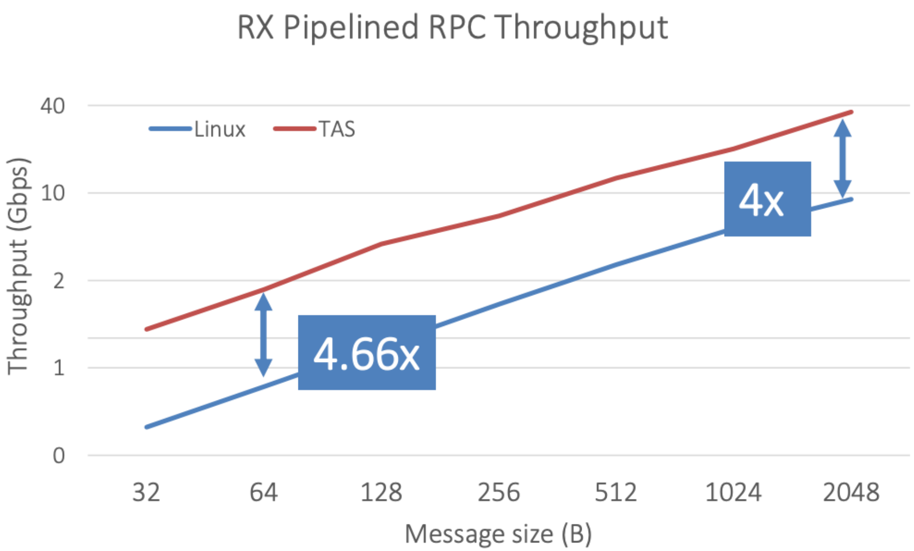
- Application runs on separate core.
- Not zero-copy.

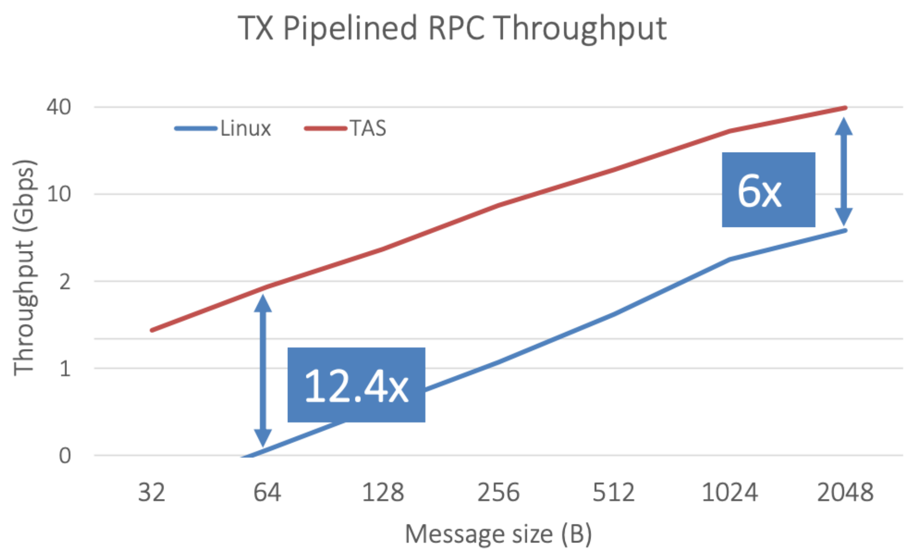
Workload Proportionality



Linux vs TAS on RPCs (1 App Core)

- Single direction RPC benchmark
- 32 RPCs per connection in flight
- 250 cycle application workload
- 64 bytes realistic small RPC

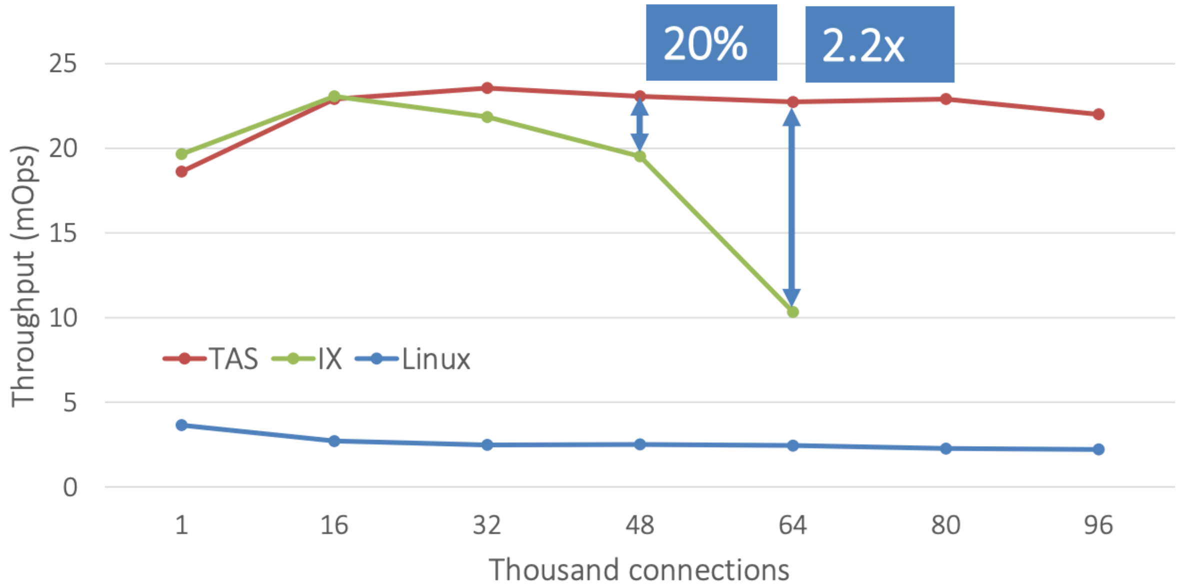




Connection Scalability

- 20 core RPC echo server
- 64B requests/responses
- Single RPC per connection

Key factor: minimized connection state



Key-value Store

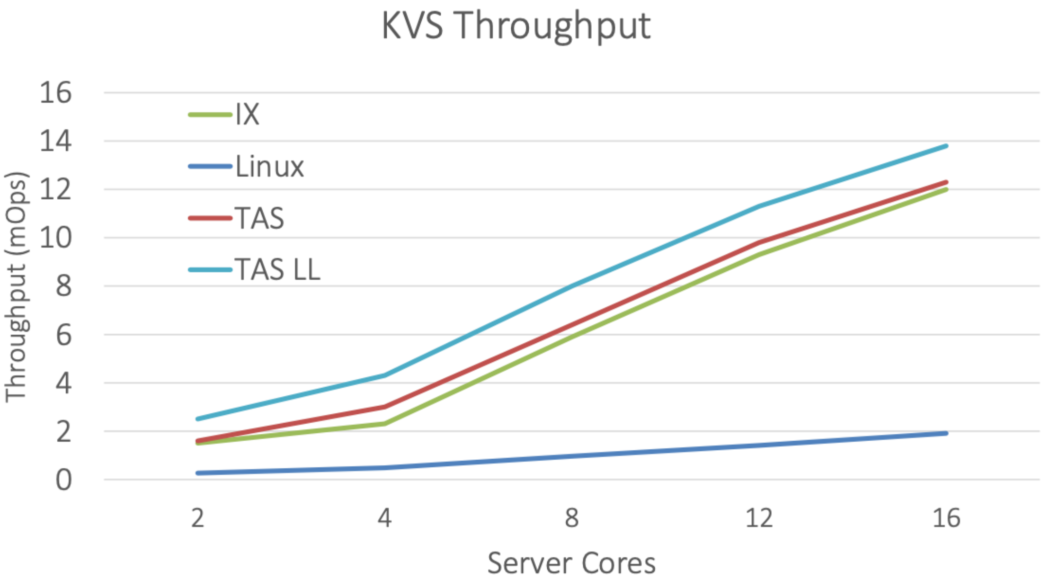
Increasing server cores with matching load (~2000 connections per core)

IX and TAS provide ~6x speedup over Linux across all cores

- TAS: 9 app cores, 7 TAS cores
- IX, Linux: 16 app/stack cores

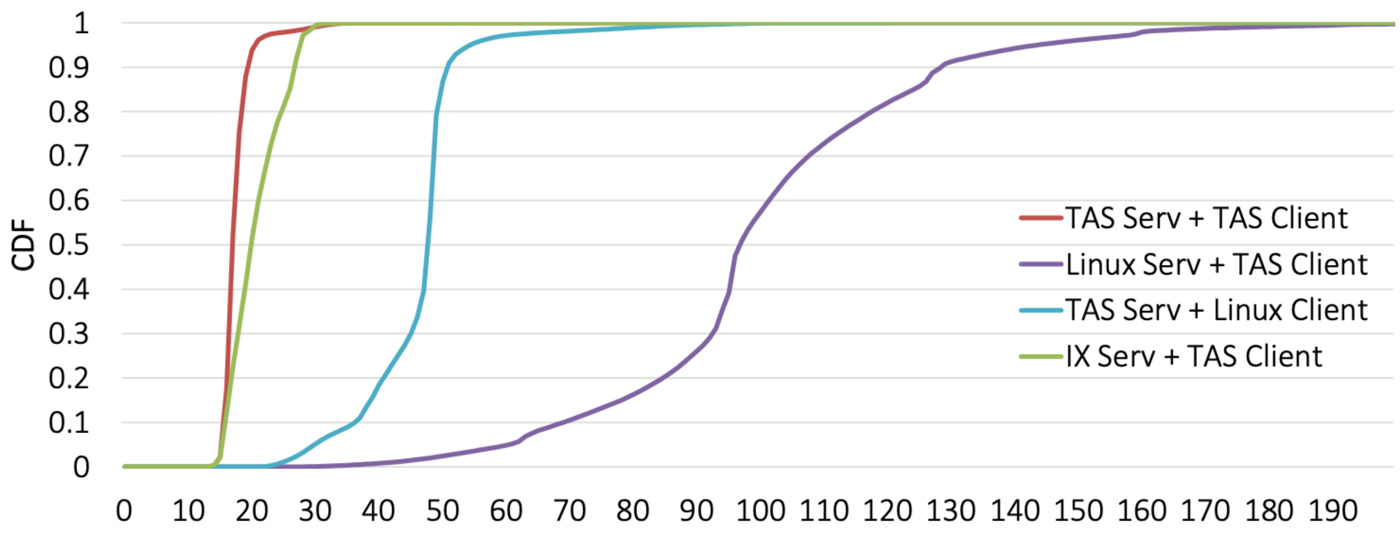
TAS has a 15-20% performance improvement over IX without sockets

- TAS: 8 app cores, 8 TAS cores



Key-value Store Latency

KVS latency measure with single application core, 15% server load



Tail Latency

IX has 50% higher latency in the 90p case

Latency is 20us (27%) higher in the 99.99p case

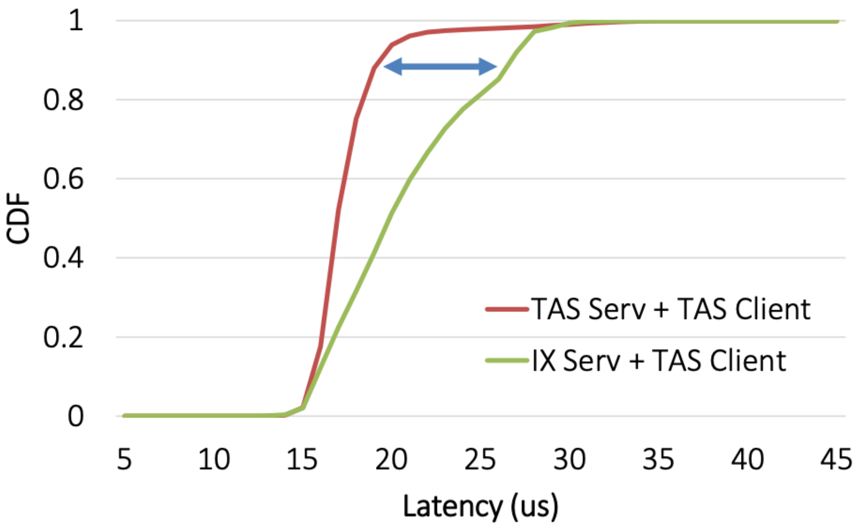
In addition, IX has a 2.3x higher maximum latency

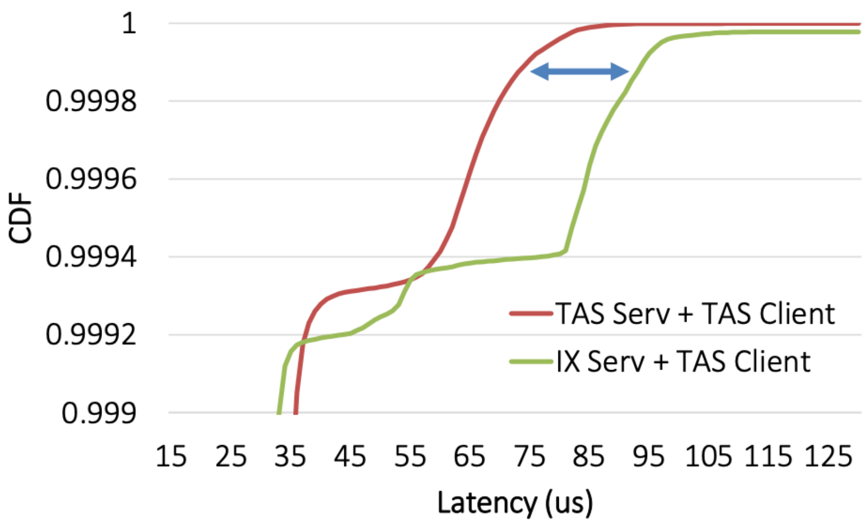
Why long IX tail?

Batching

TAS Max: 122us

IX Max: 280us





Your thoughts…