TAS: TCP Acceleration as an OS Service

Antoine Kaufmann[1], **Tim Stamler[2]**, Simon Peter[2],
Naveen Kr. Sharma[3], Arvind Krishnamurthy[3], Thomas Anderson[3]

MPI-SWS[1] *The University of Texas at Austin[2]* *University of Washington[3]*

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?

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?

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?

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*

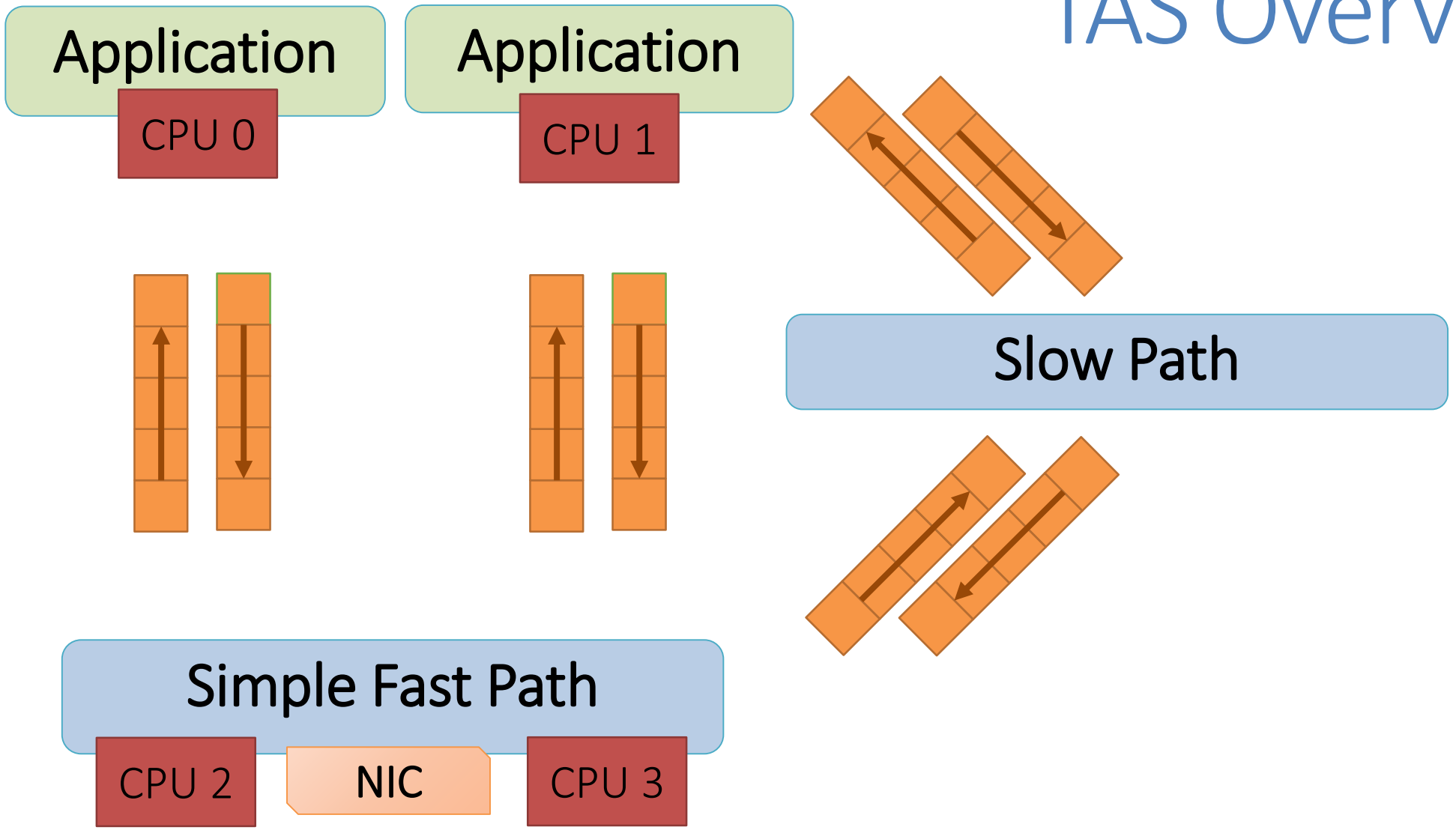
How does TAS fix it?

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



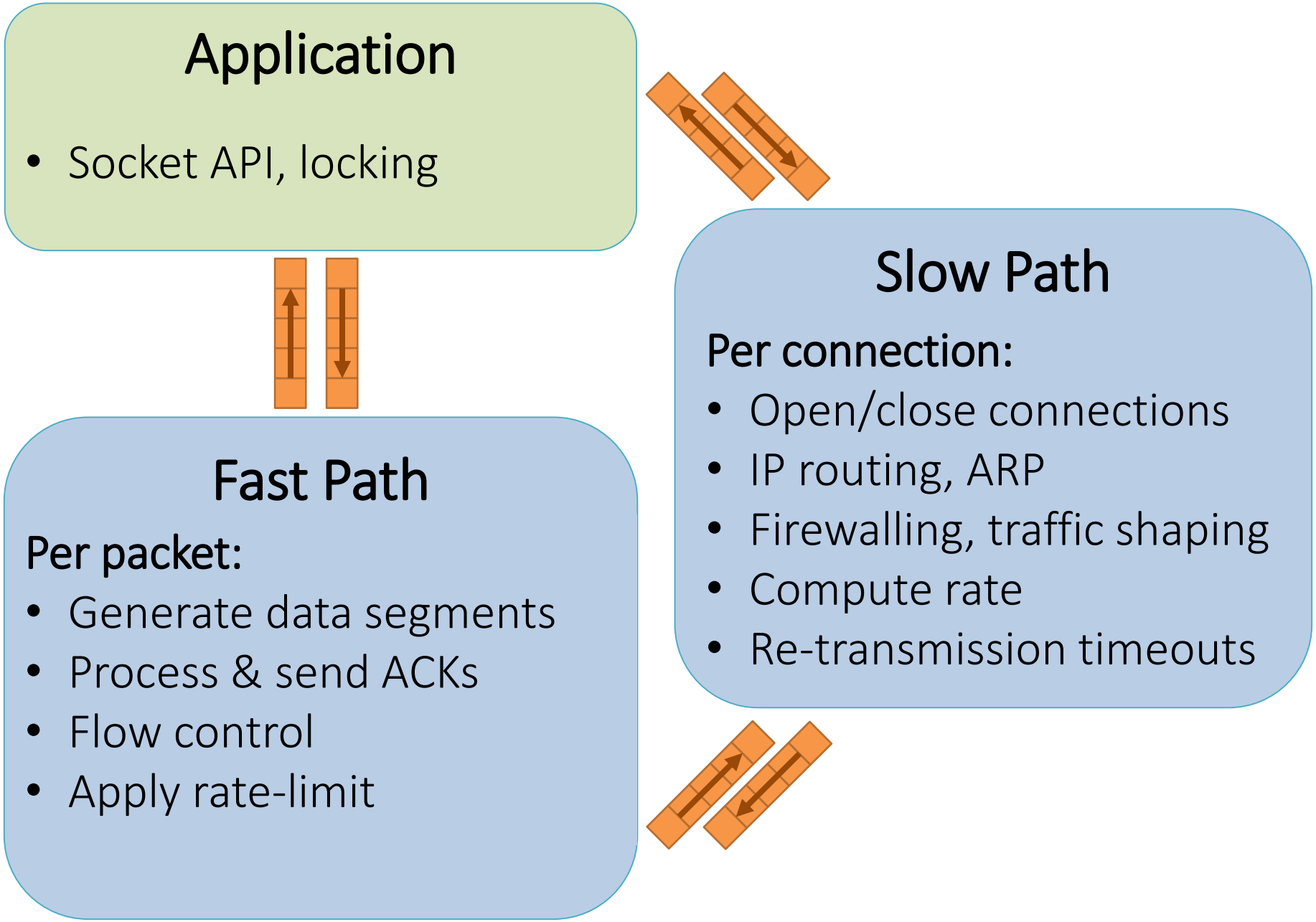
Dividing Functionality

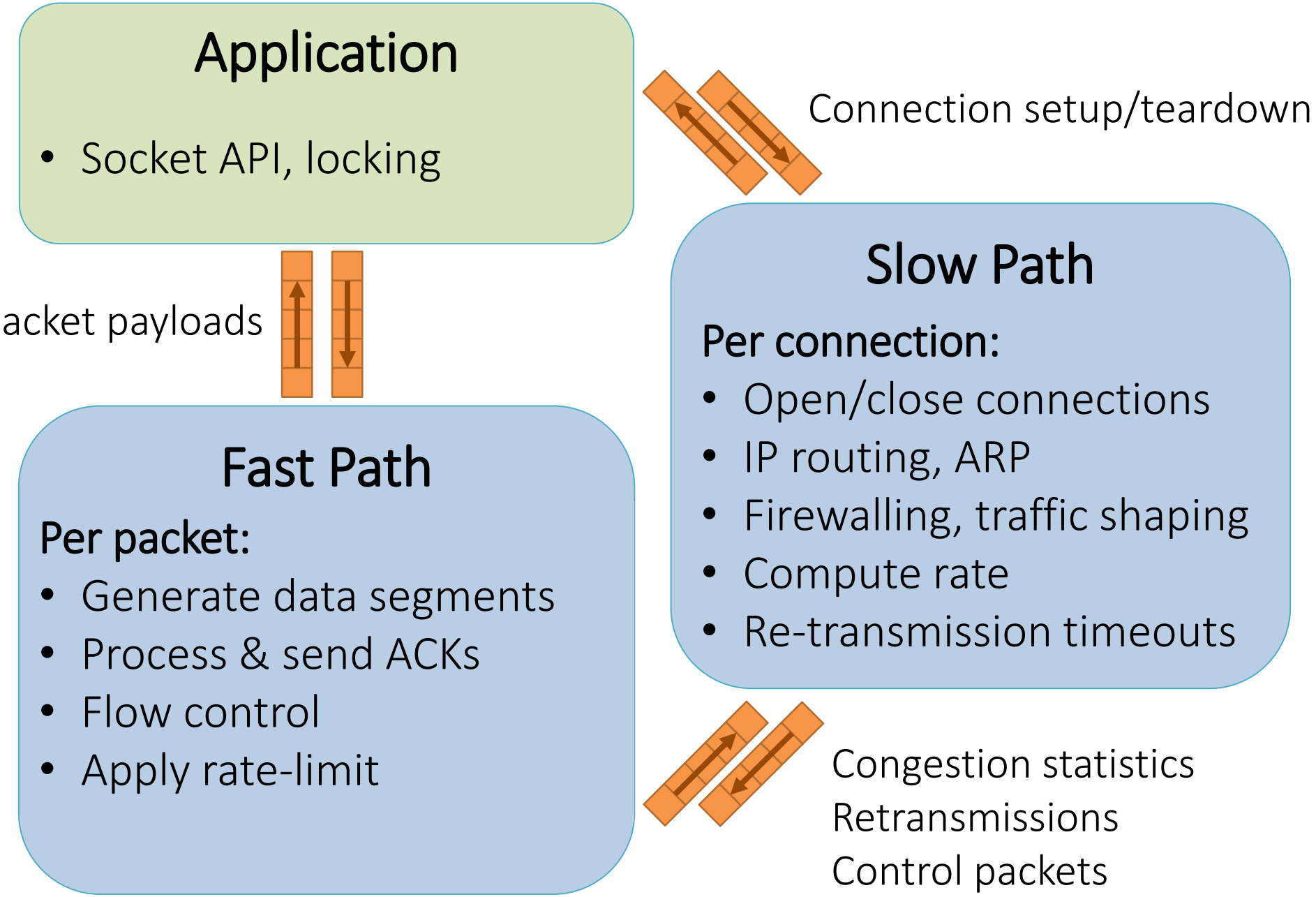
Linux Kernel TCP Stack

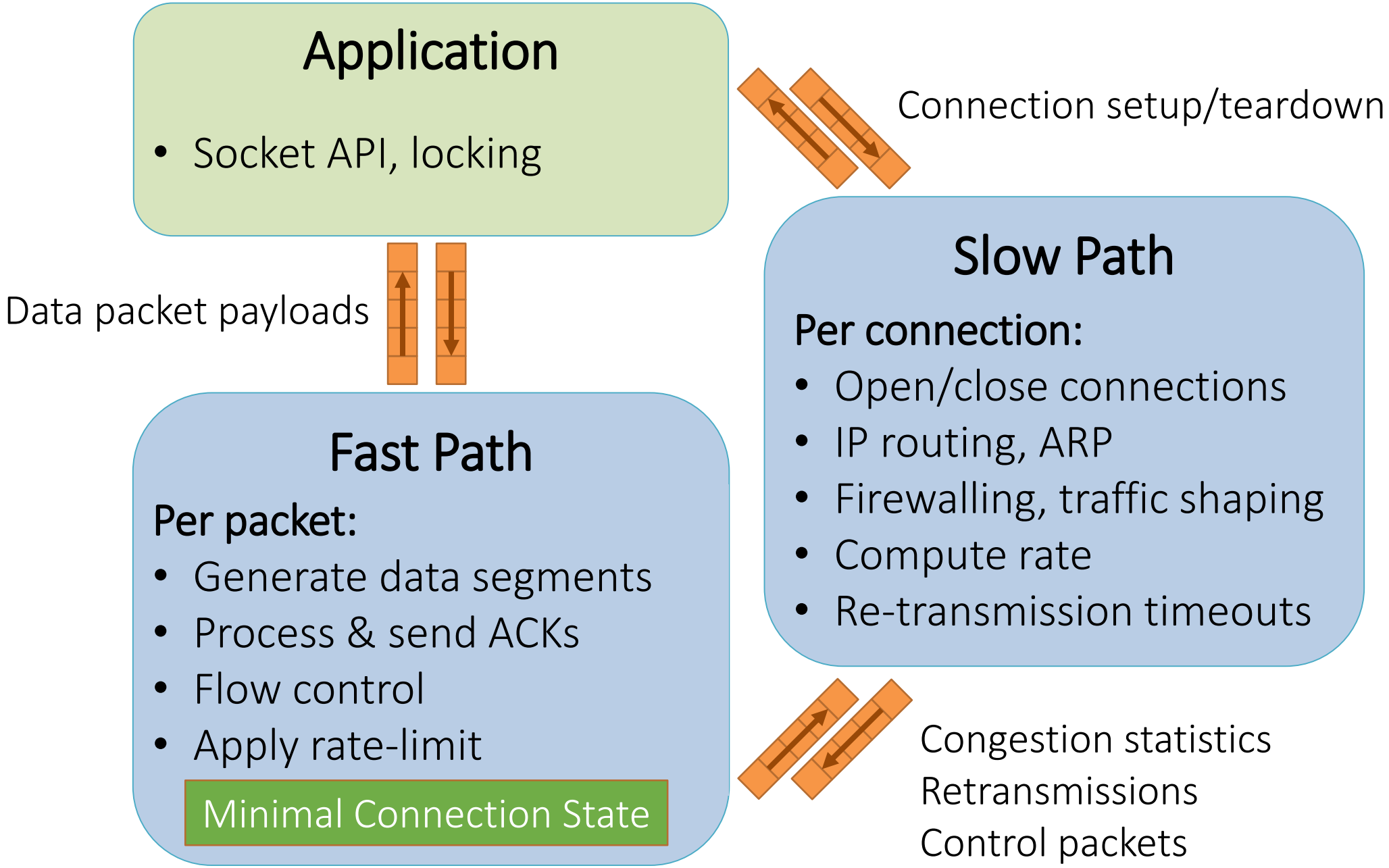
- Open/close connections

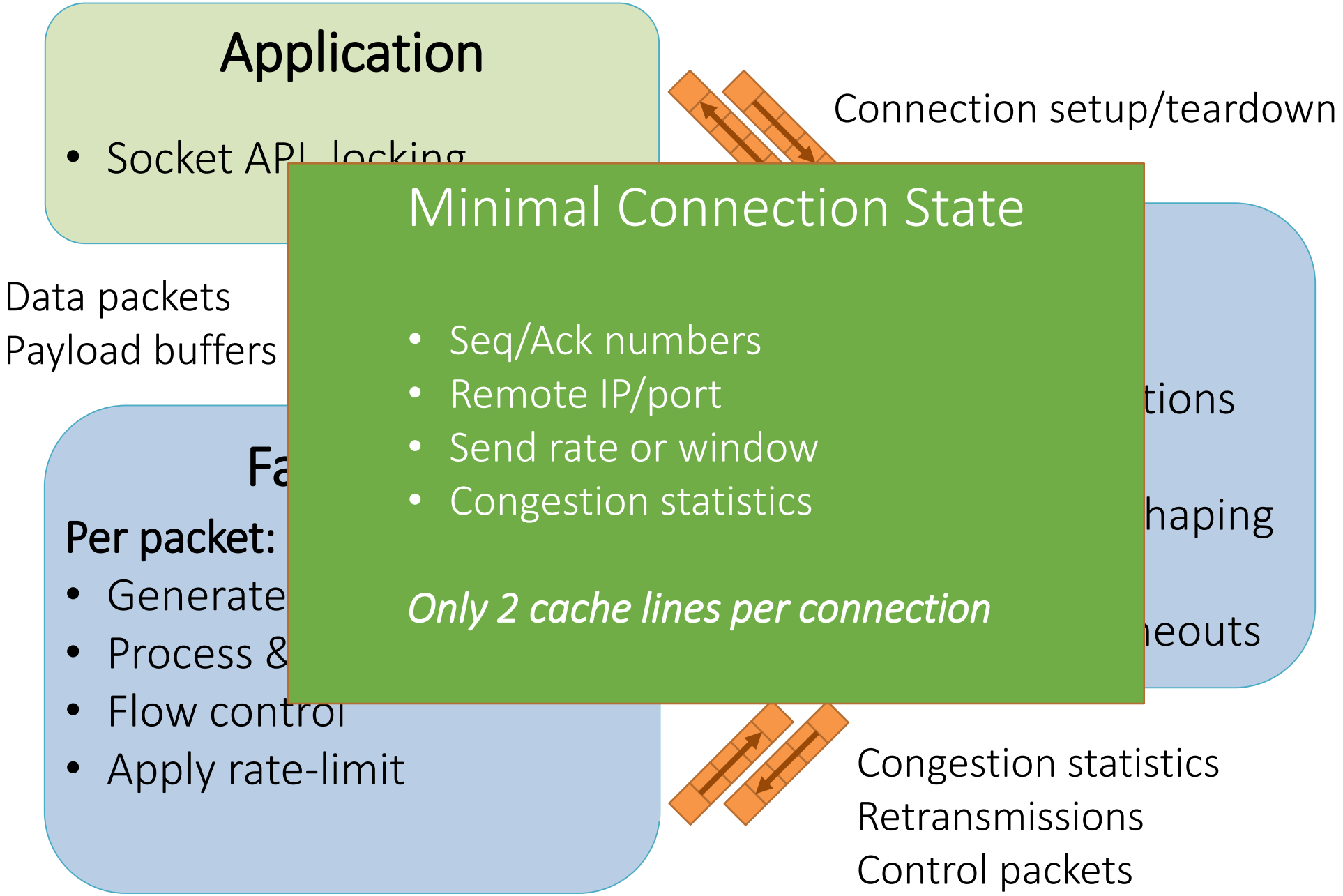
Per packet:

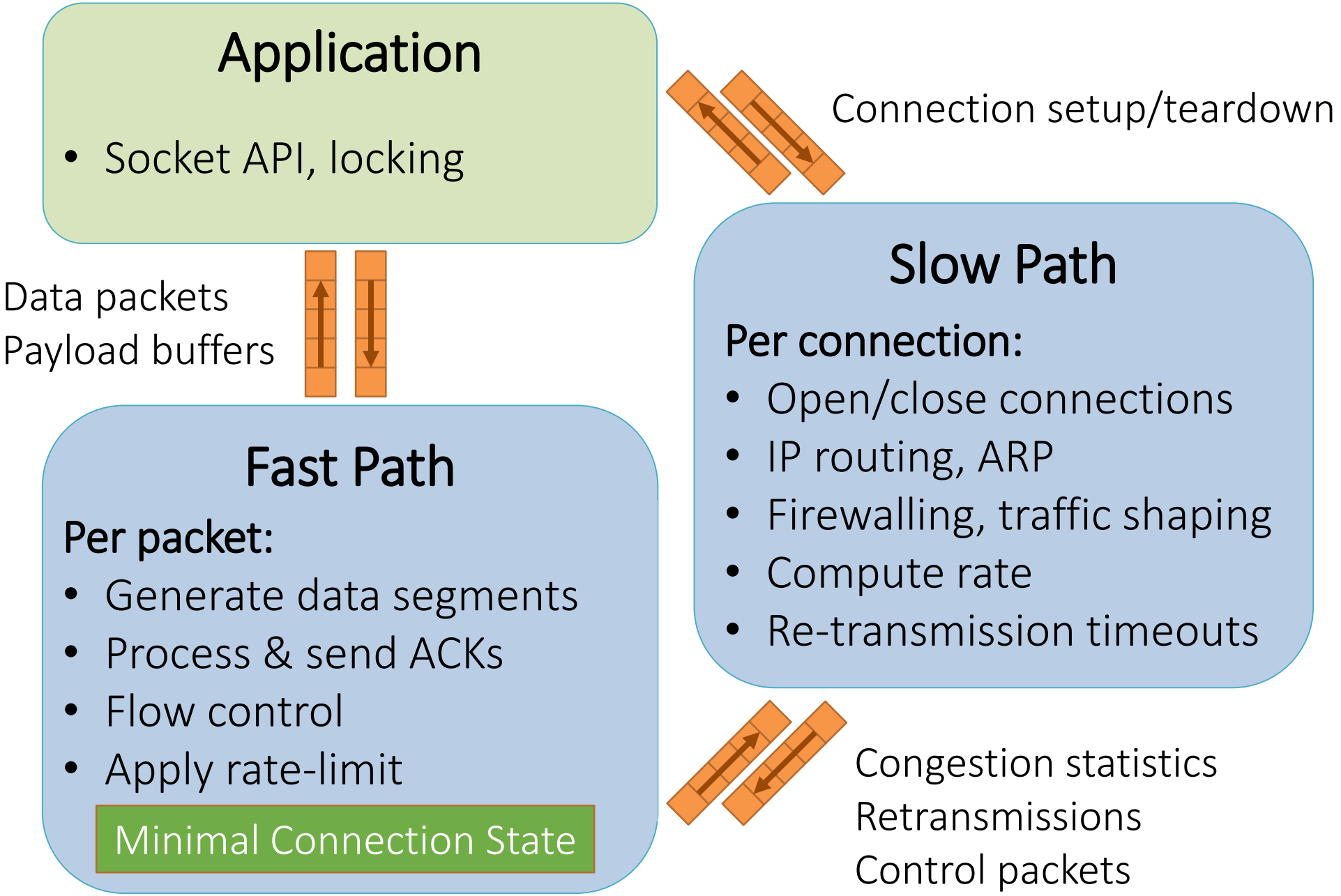
- Socket API, locking
- IP routing, ARP
- Firewalling, traffic shaping
- Generate data segments
- Congestion control
- Flow control
- Process & send ACKs
- Re-transmission timeouts

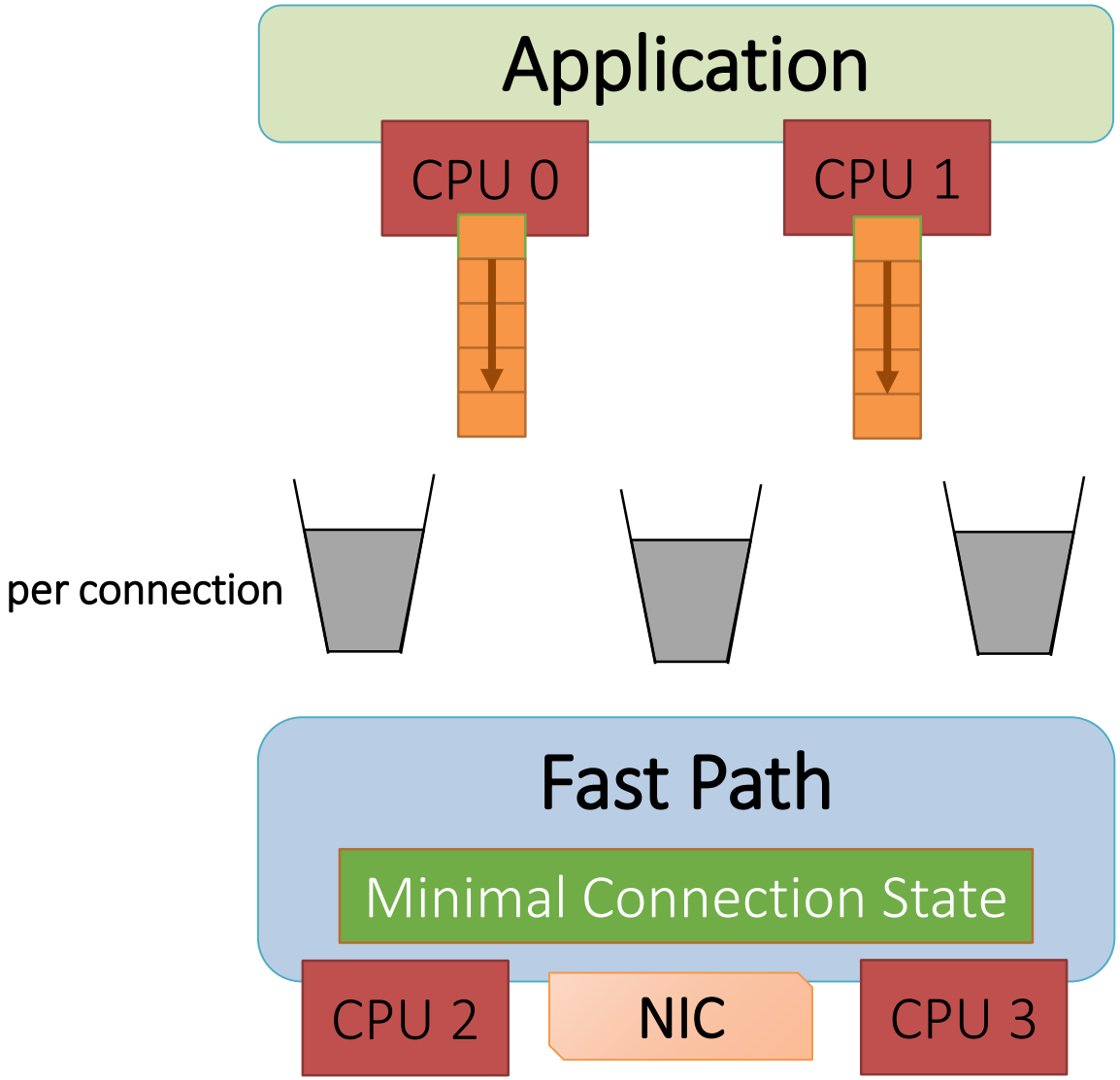

Application
• Socket API, locking
Slow Path
Per connection:
• Open/close connections
• IP routing, ARP
• Firewalling, traffic shaping
• Compute rate
• Re-transmission timeouts
Fast Path
Per packet:
• Generate data segments
• Process & send ACKs
• Flow control
• Apply rate-limit


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


Application
• Socket API, locking
Connection setup/teardown
Data packets
Payload buffers
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



Congestion Control

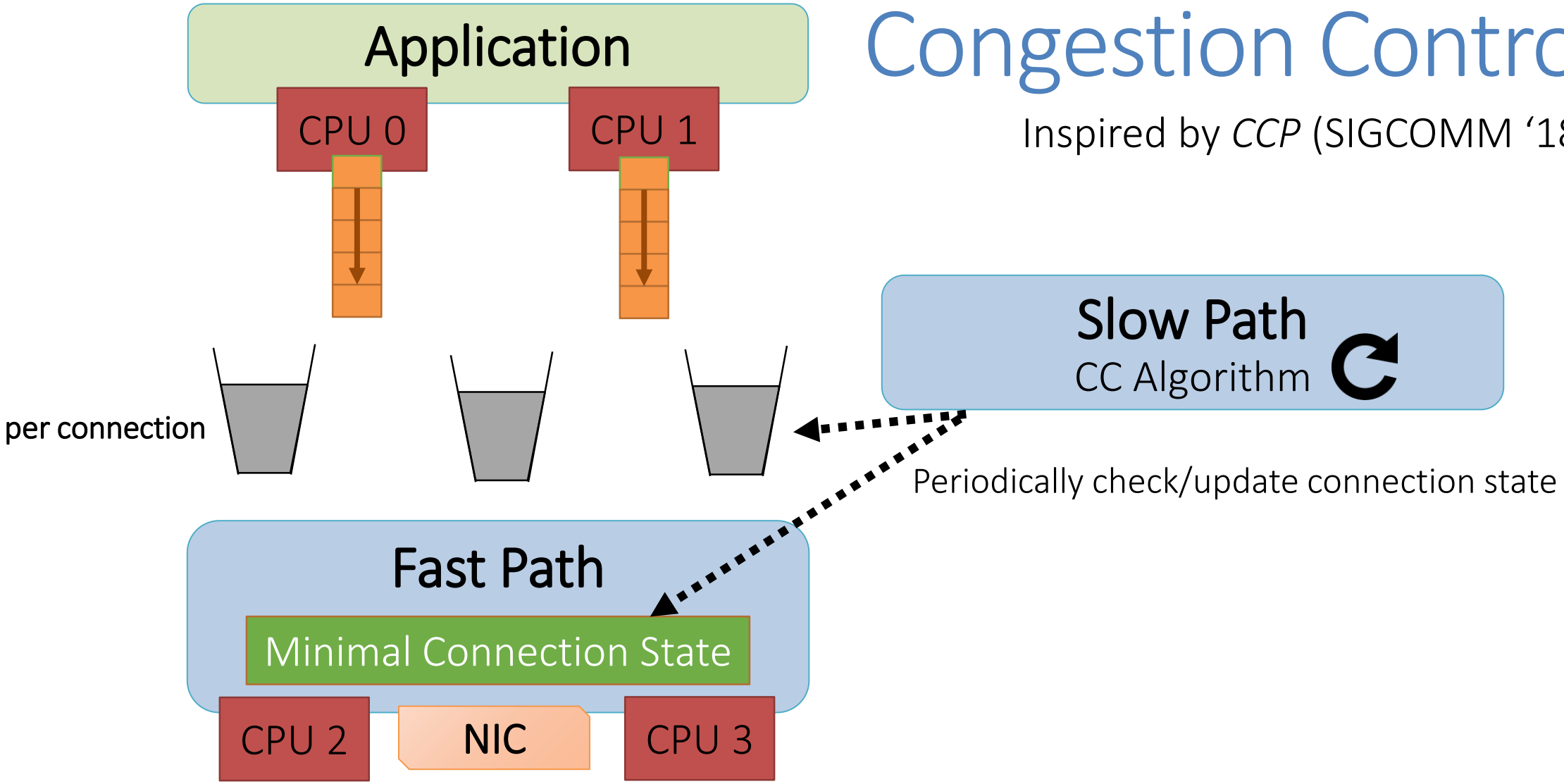
Inspired by *CCP* (SIGCOMM '18)

Slow Path
CC Algorithm

Periodically check/update connection state

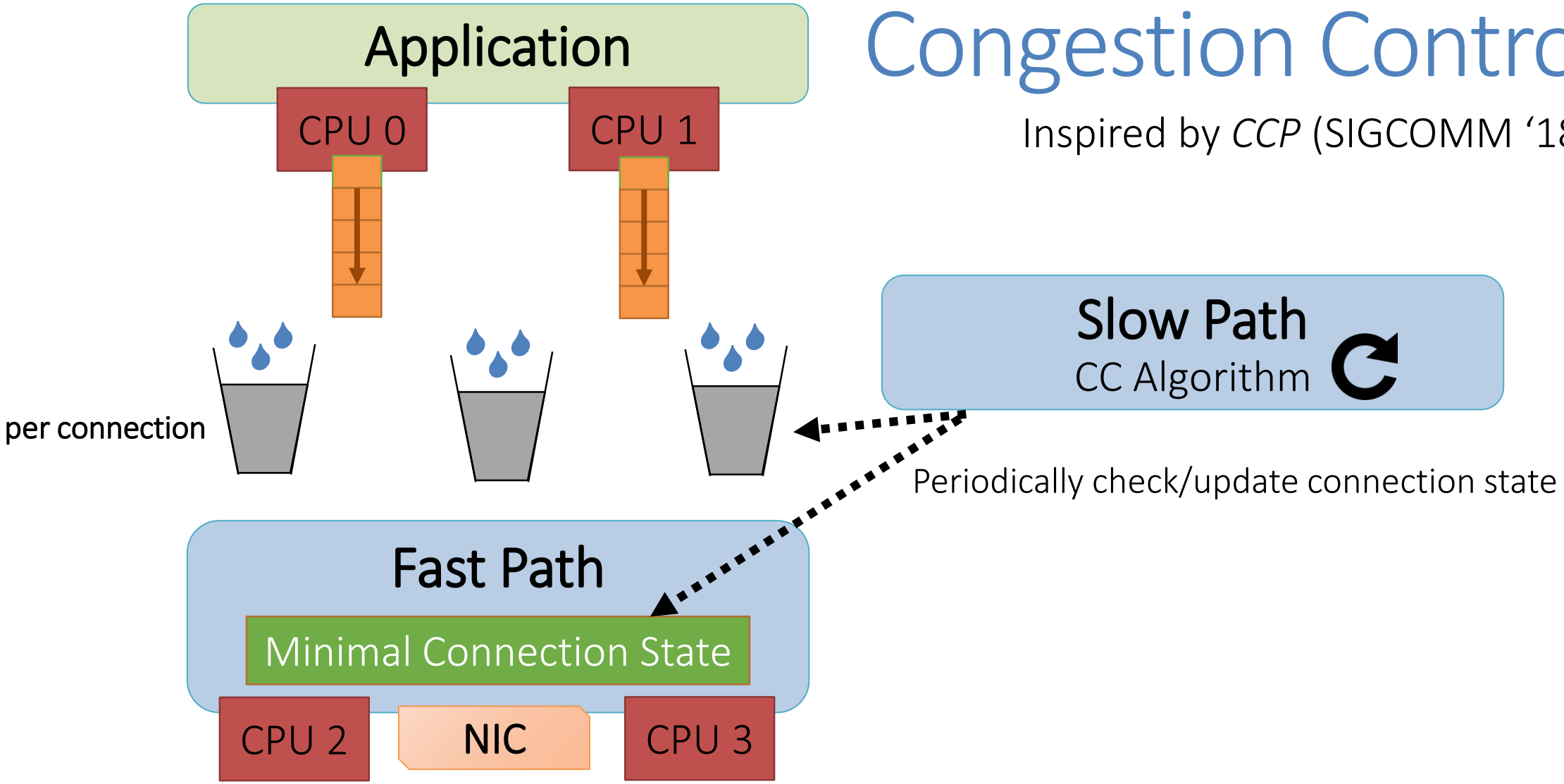
Congestion Control

Inspired by *CCP* (SIGCOMM '18)



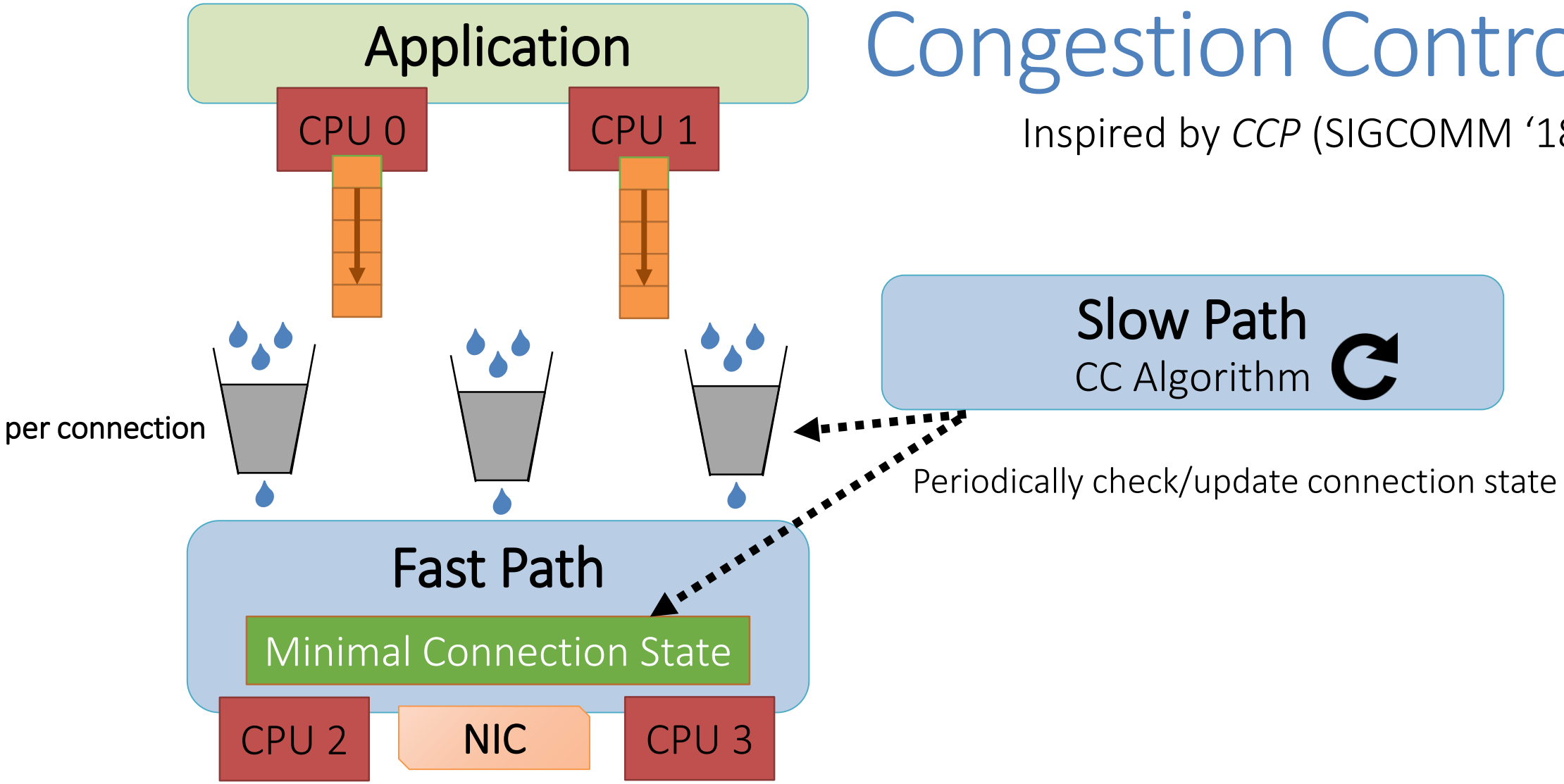
Congestion Control

Inspired by *CCP* (SIGCOMM '18)



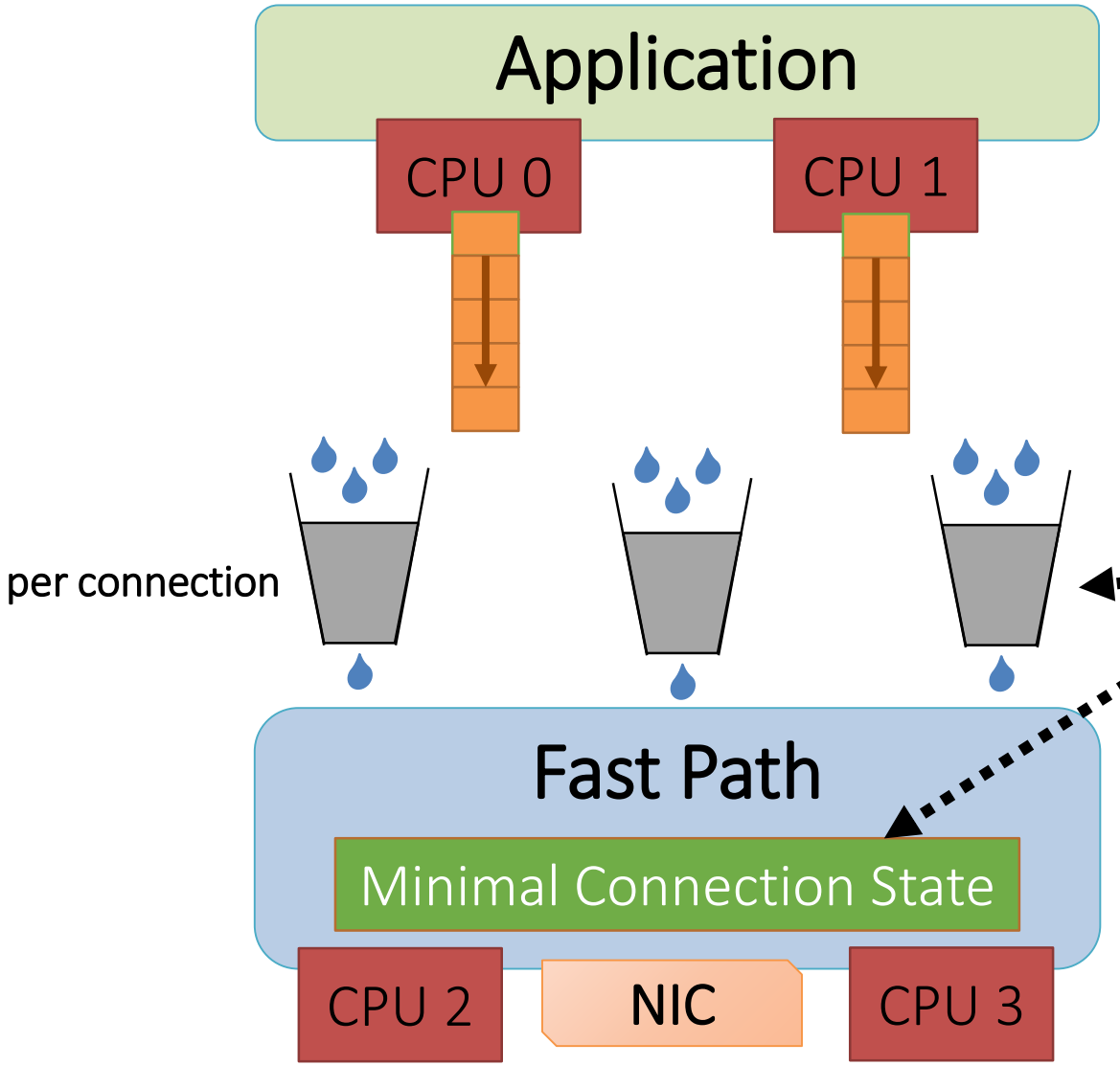
Congestion Control

Inspired by *CCP* (SIGCOMM '18)



Congestion Control

Inspired by *CCP* (SIGCOMM '18)

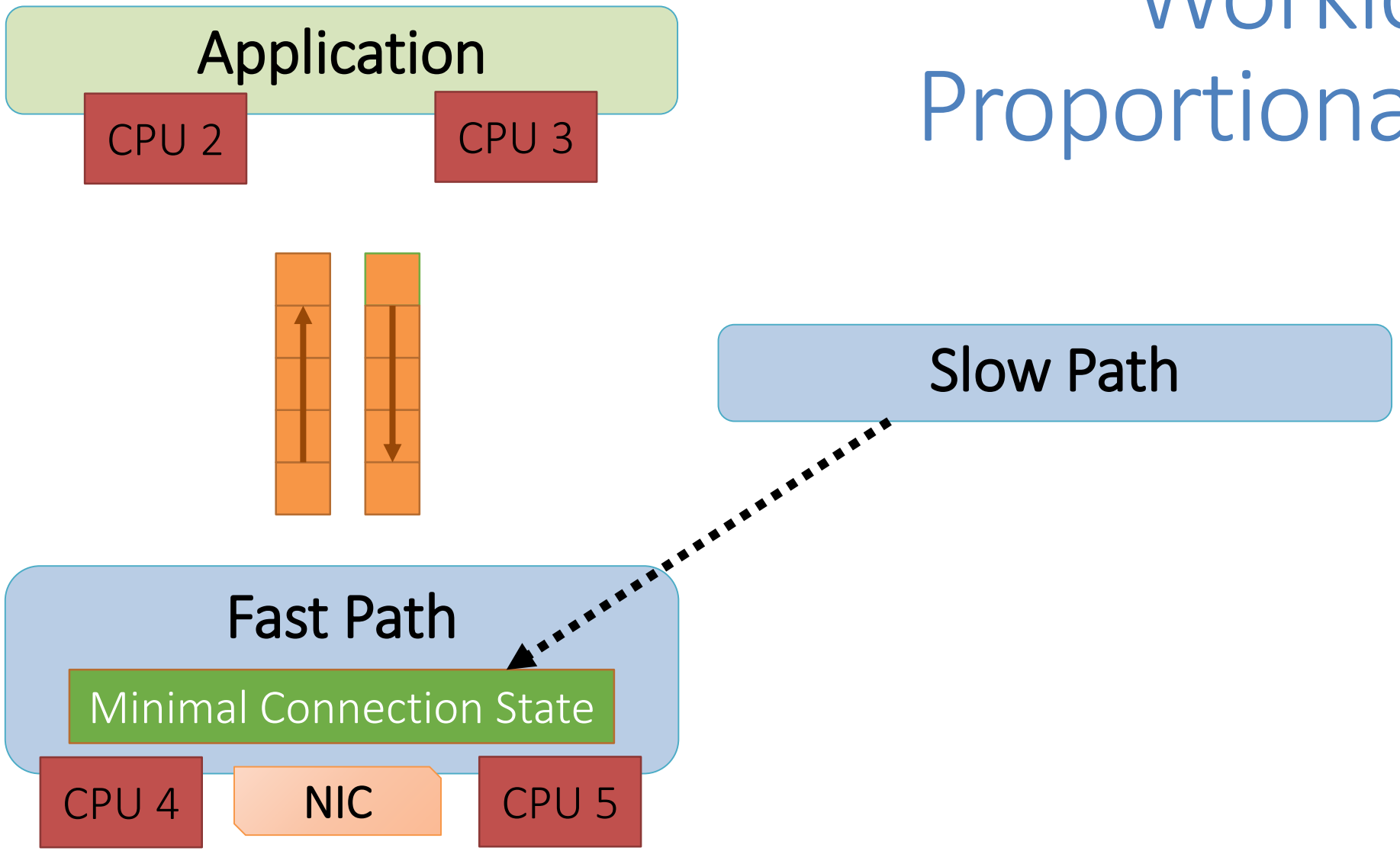


Slow Path
CC Algorithm

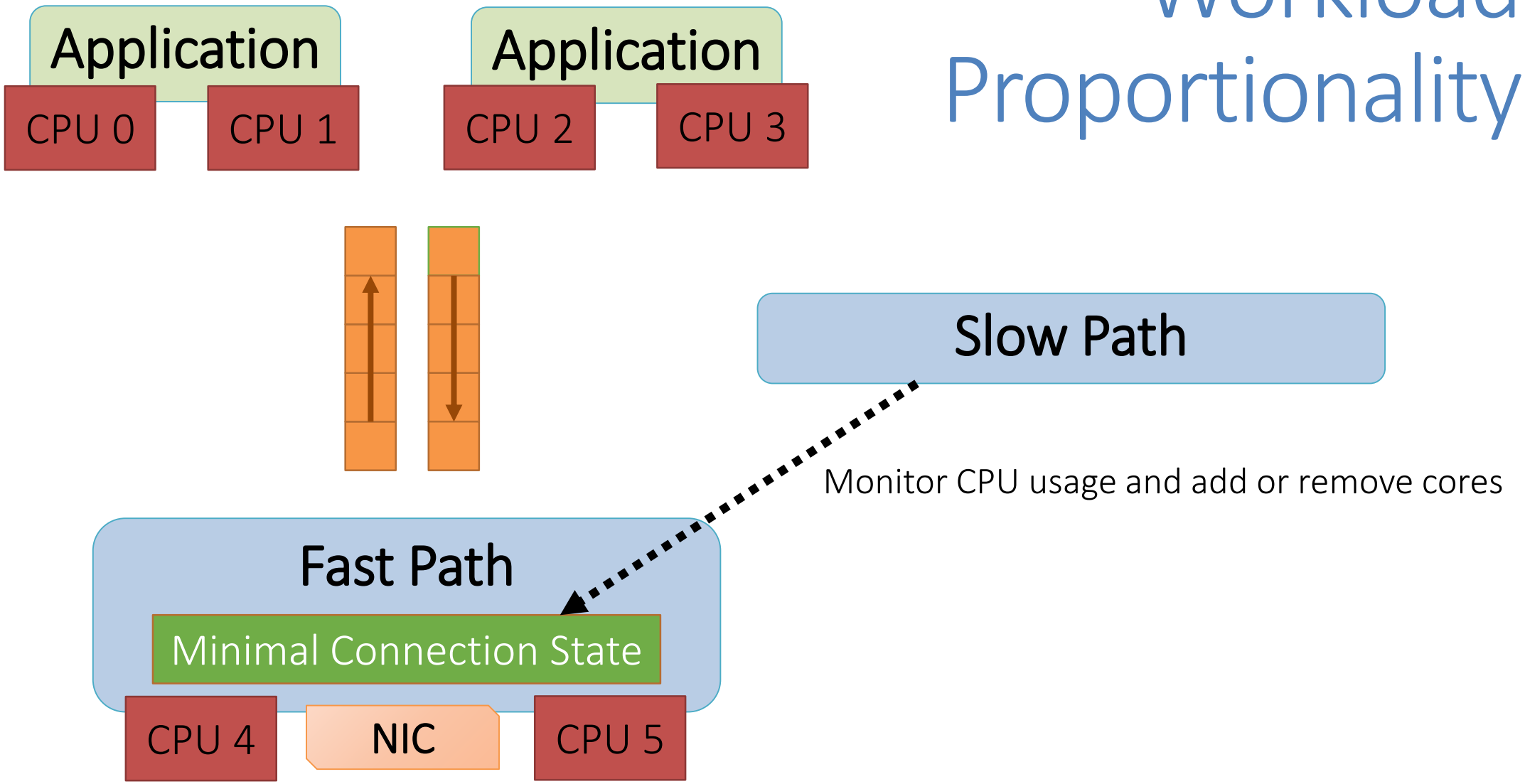
Periodically check/update connection state

Many CC algorithms can be implemented (described in paper)

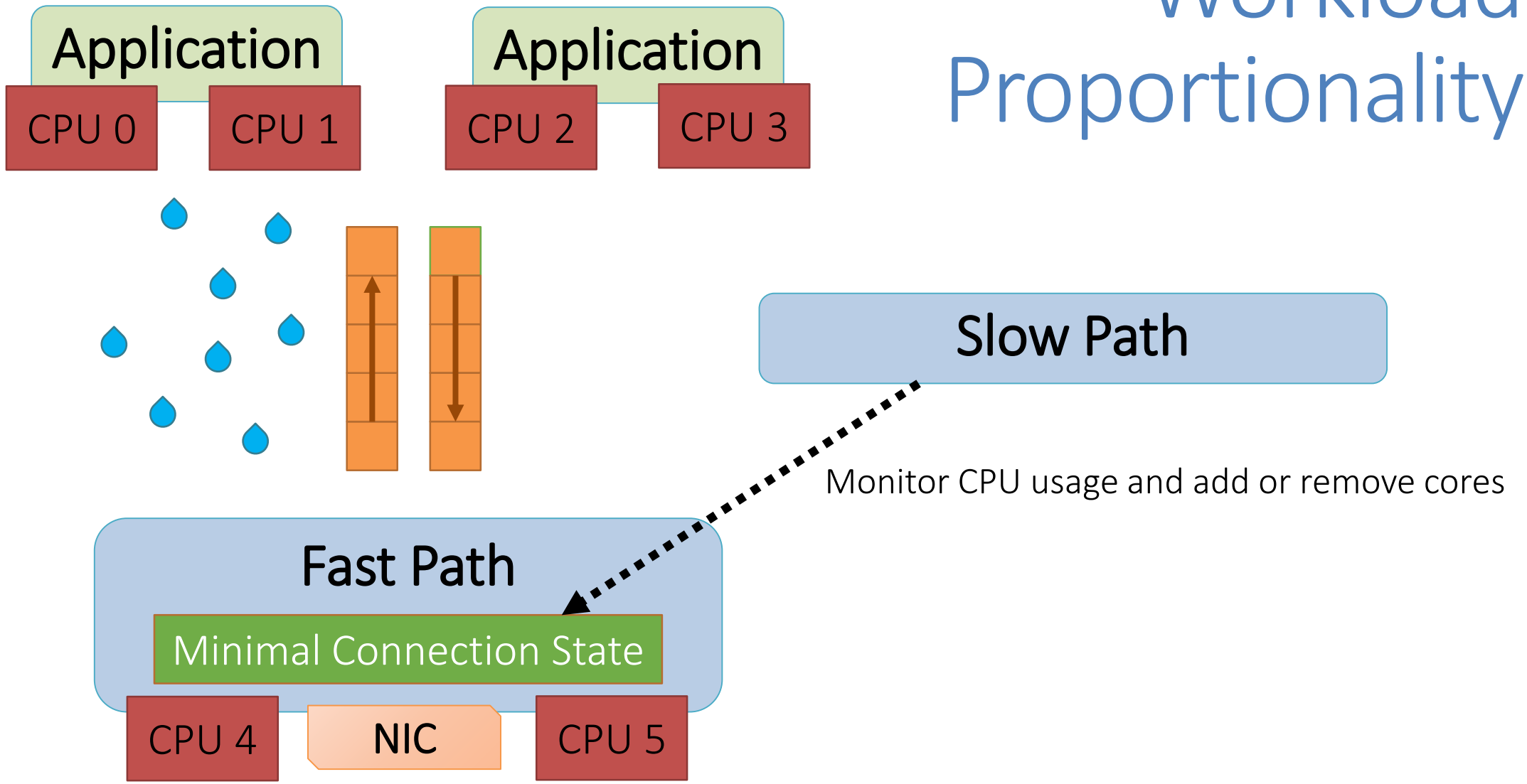
Workload Proportionality



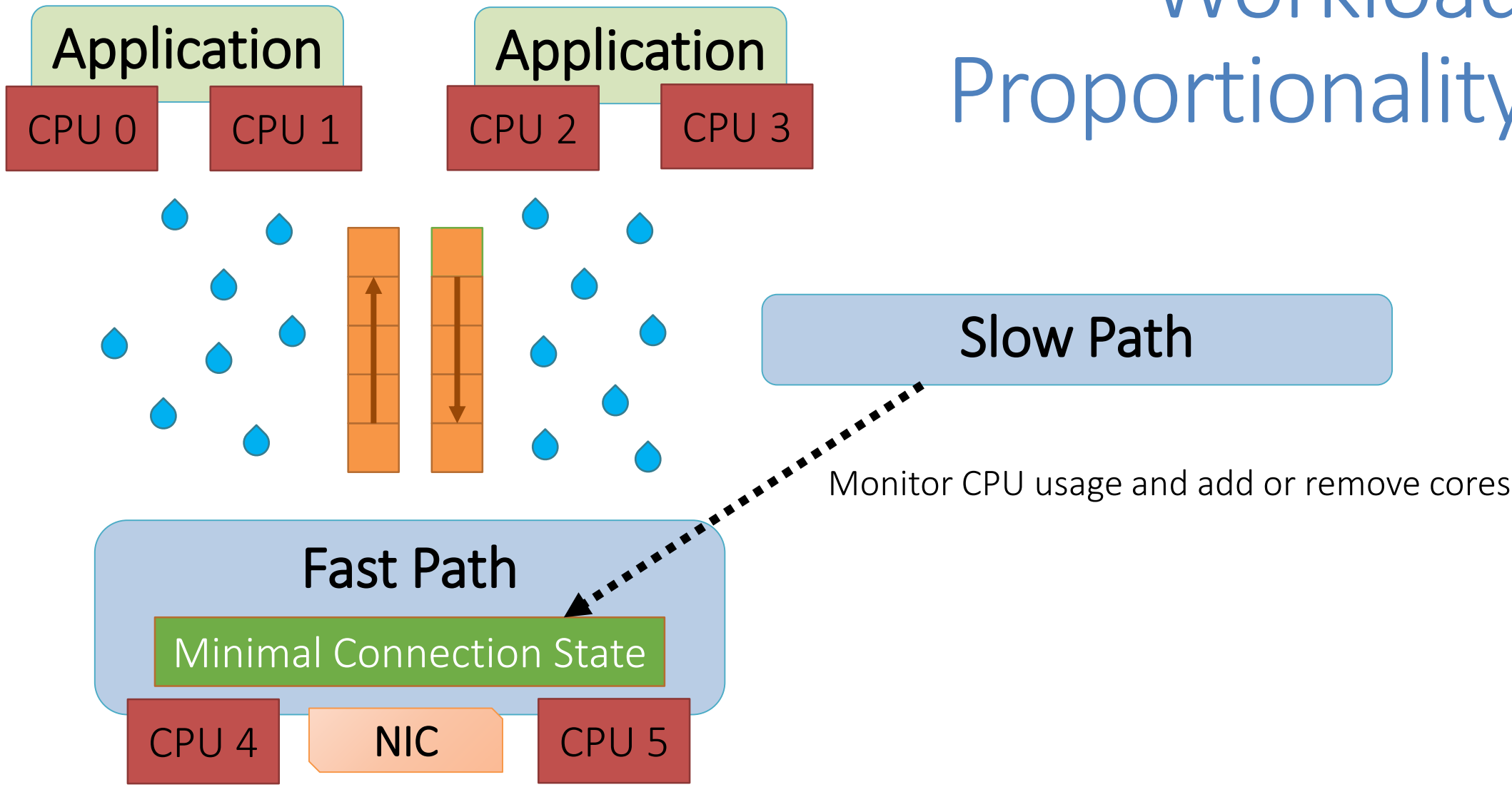
Workload Proportionality



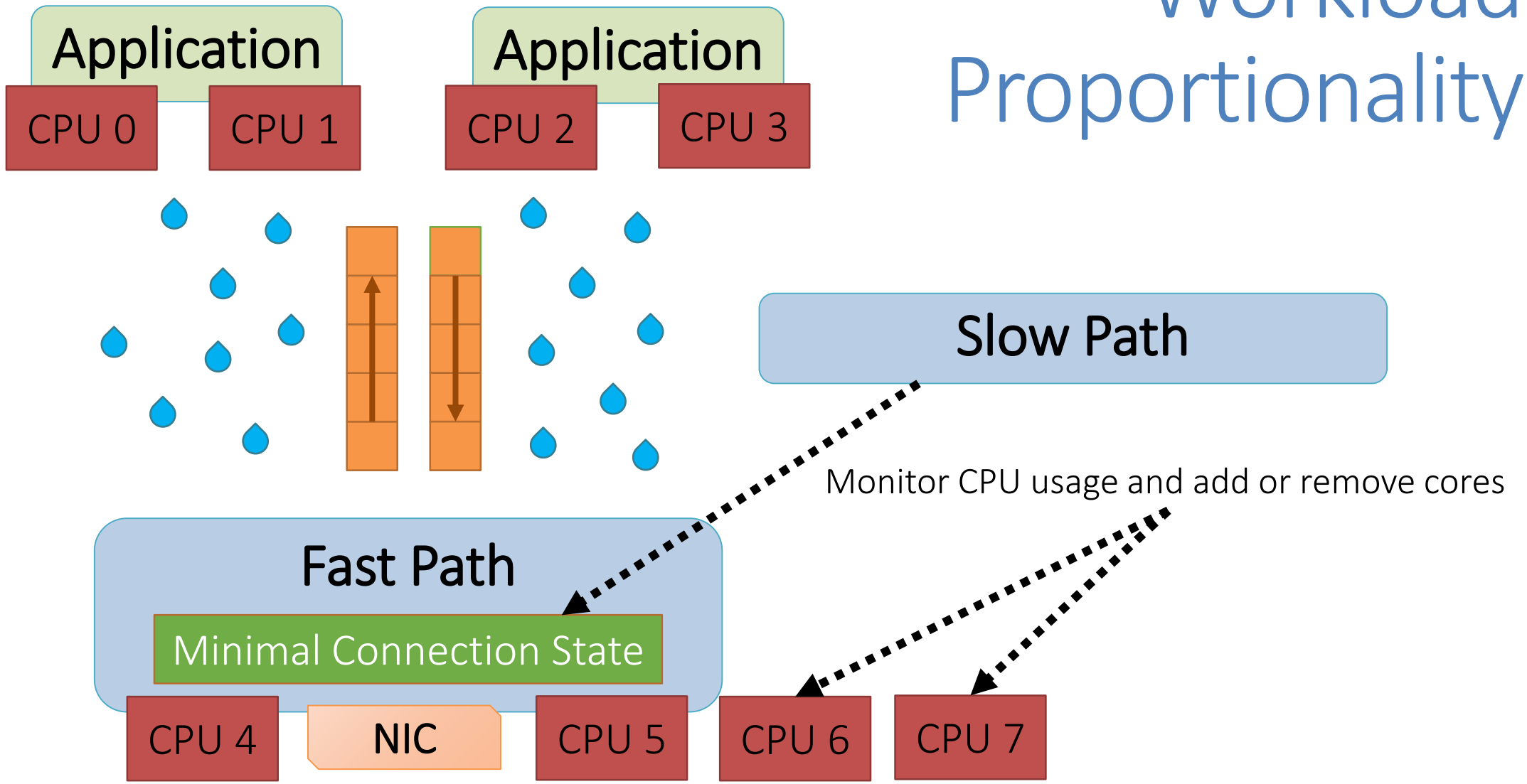
Workload Proportionality



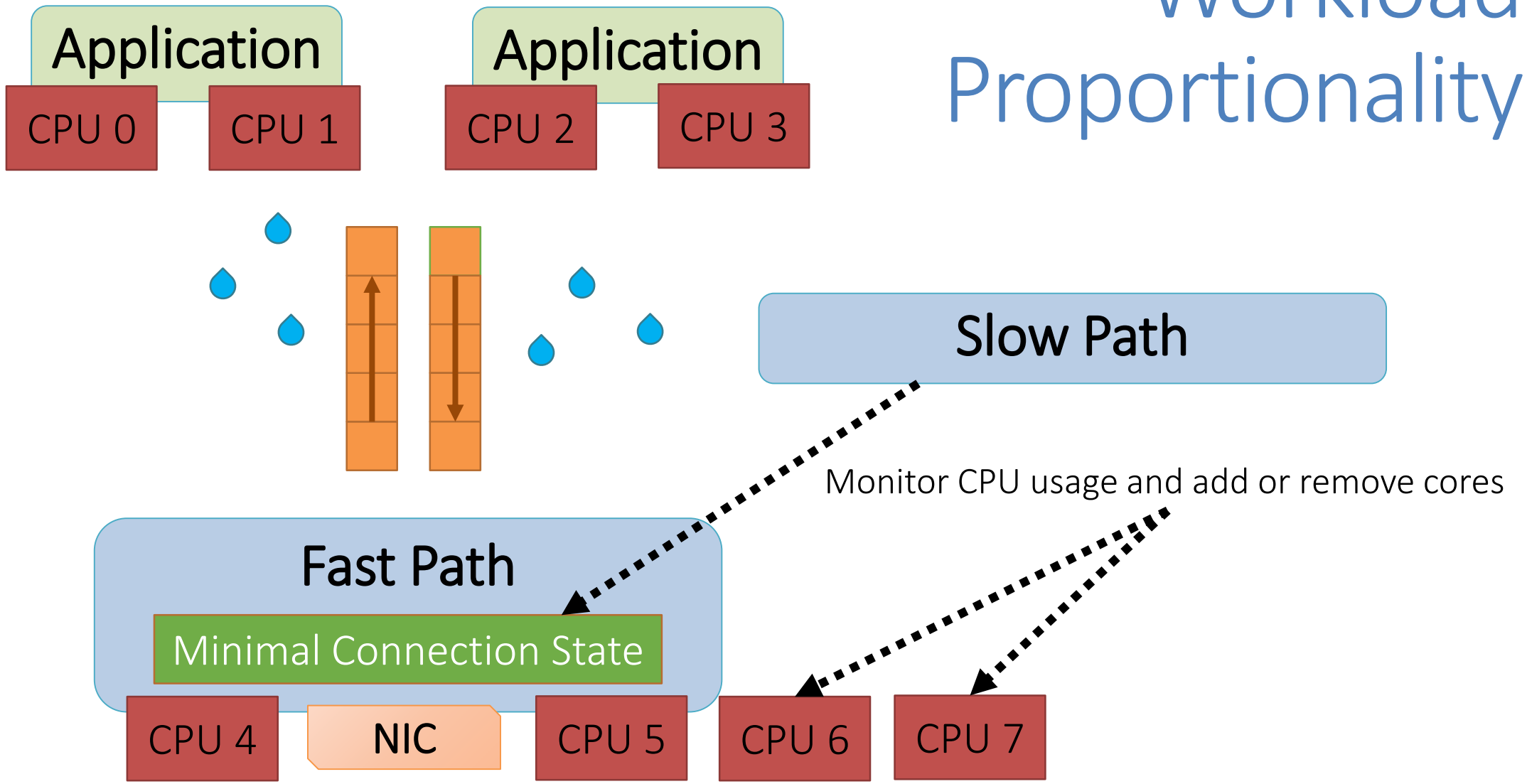
Workload Proportionality



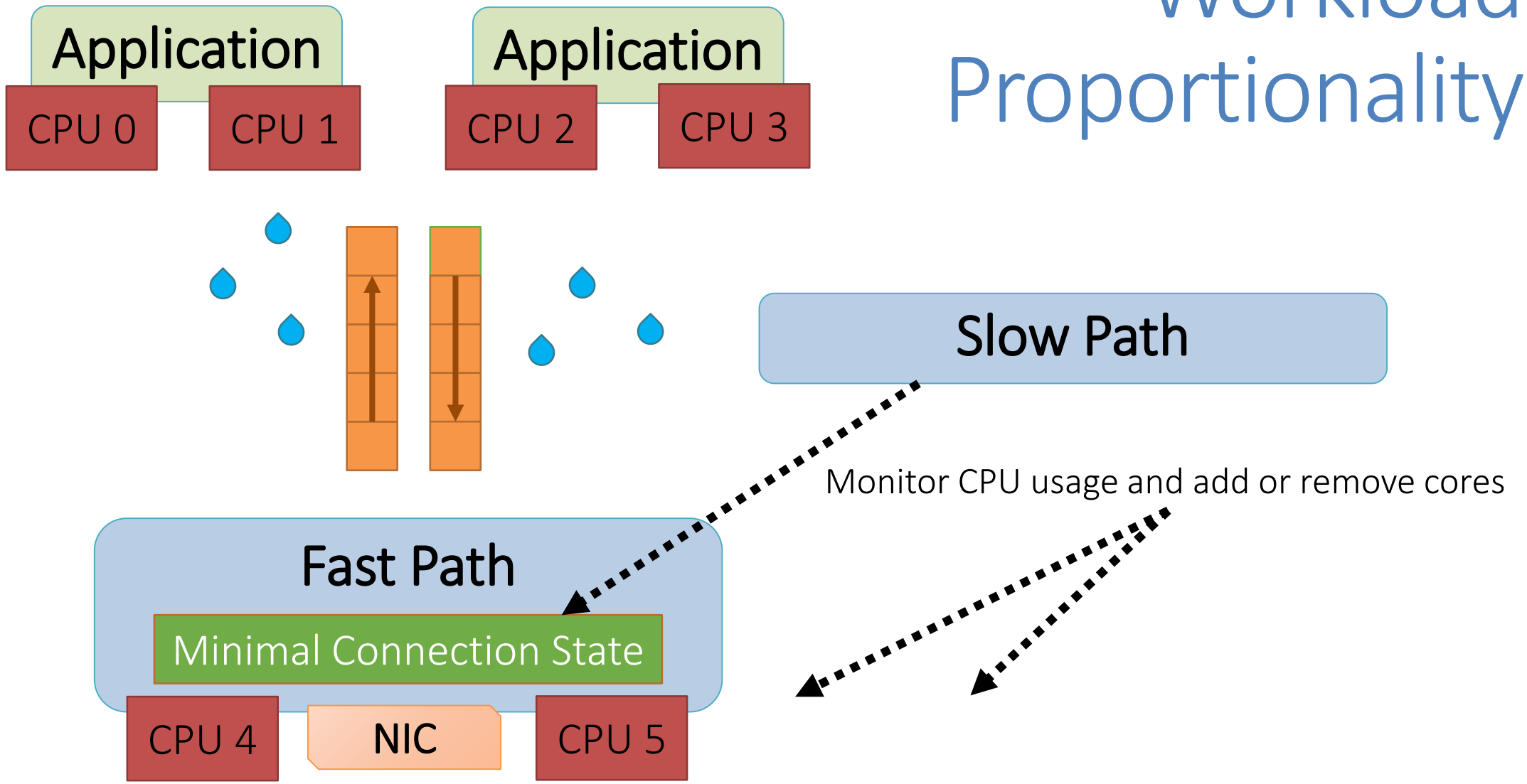
Workload Proportionality



Workload Proportionality



Workload Proportionality



Evaluation

Evaluation Questions

What is our throughput, latency, and scalability for RPCs?

Do real applications scale with # of cores and have low tail latency?

Do we distribute throughput fairly under network congestion?

(See paper for more in-depth analysis)

Systems for Comparison

We evaluate TAS against 3 other systems:

1. Linux
   a) Full kernel, trusted congestion control
   b) Sockets interface
2. mTCP (not in this talk, see paper)
   a) Pure kernel bypass approach, ***untrusted congestion control***
3. IX
   a) Replace Linux with optimized data path, run in privileged mode
   b) Uses ***batching*** to reduce overhead
   c) No sockets interface
   d) Requires kernel modifications

Experimental Setup

Intel Xeon Platinum 8160 CPU 24 cores @ 2.10GHz

196GB of RAM

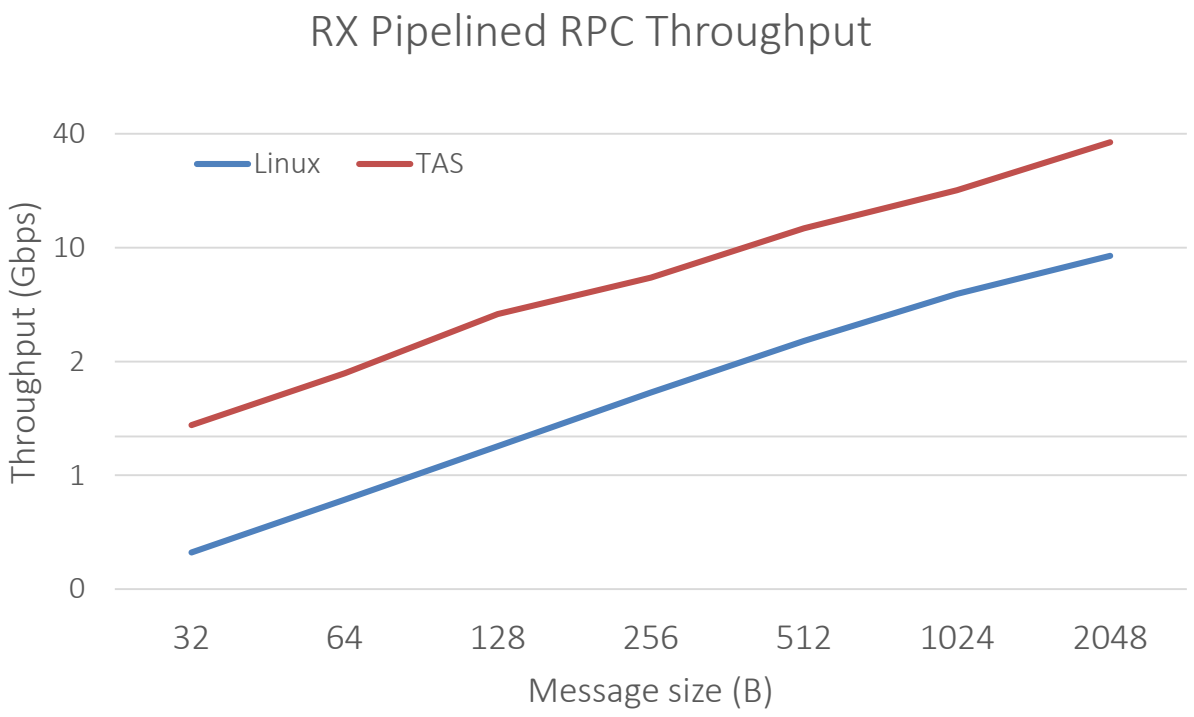
Intel XL710 40Gb Ethernet Adapter

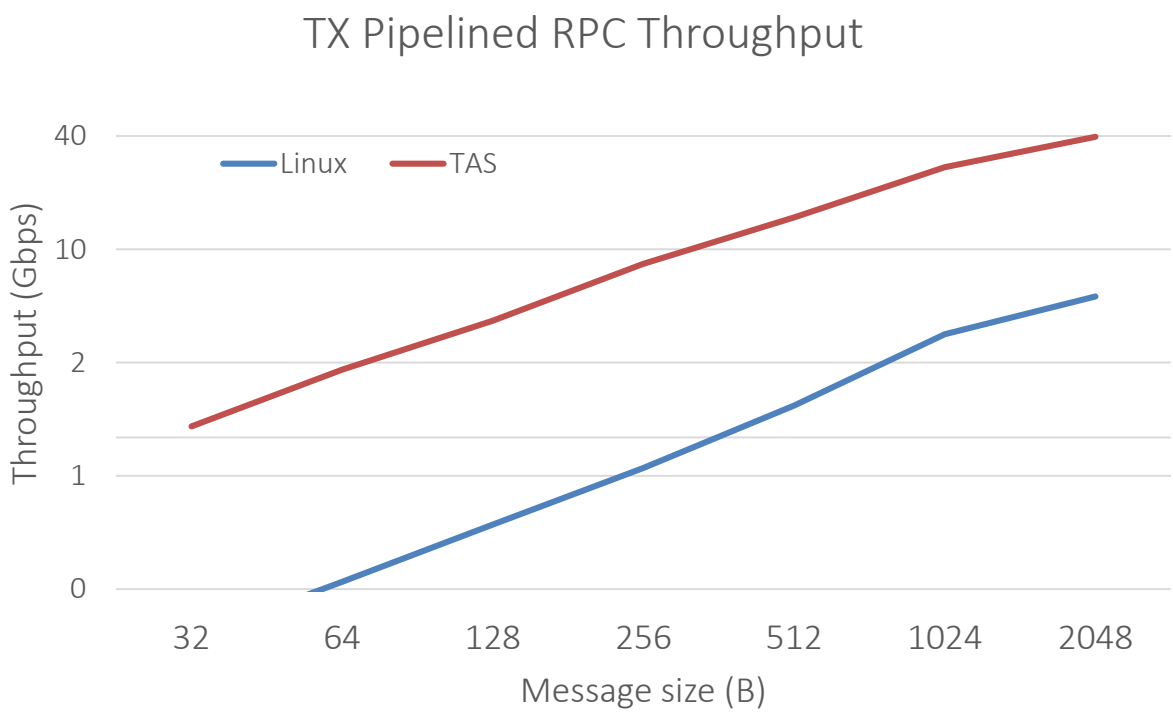
Benchmarks:

- Single direction RPC benchmark
- RPC echo server
- A scalable key-value store
- Connection throughput fairness under congestion

Linux vs TAS on RPCs (1 App Core)

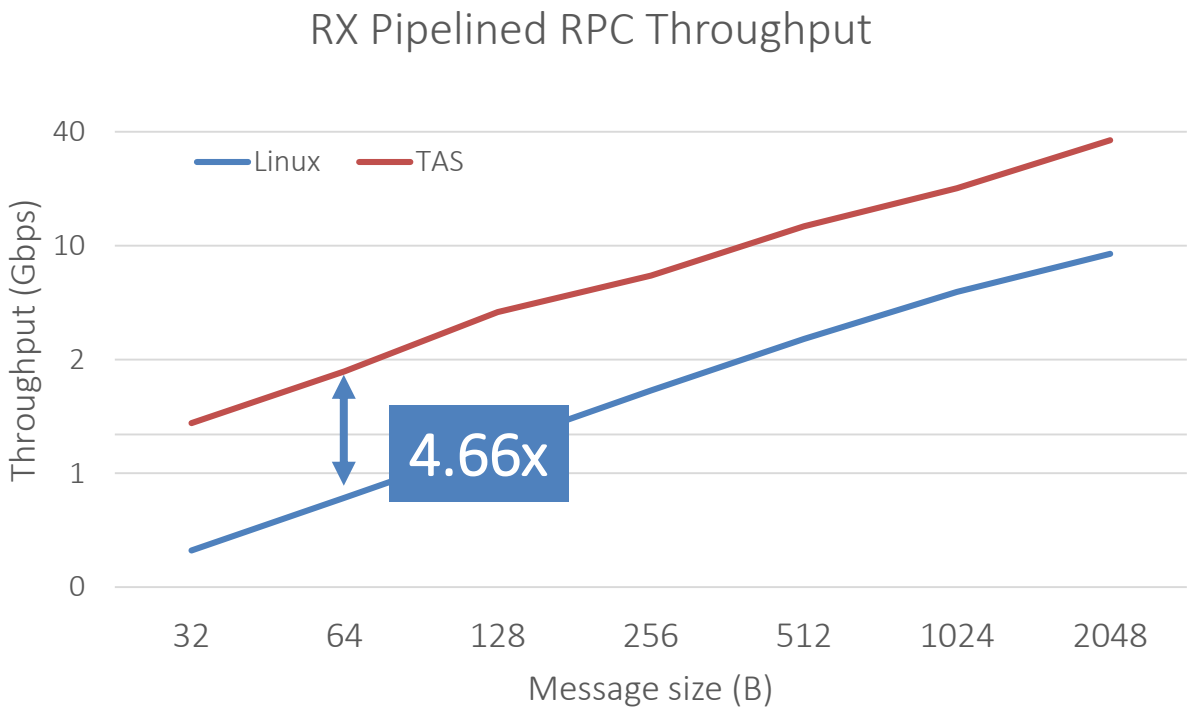
- Single direction RPC benchmark
- 32 RPCs per connection in flight
- 250 cycle application workload
- 64 bytes realistic small RPC

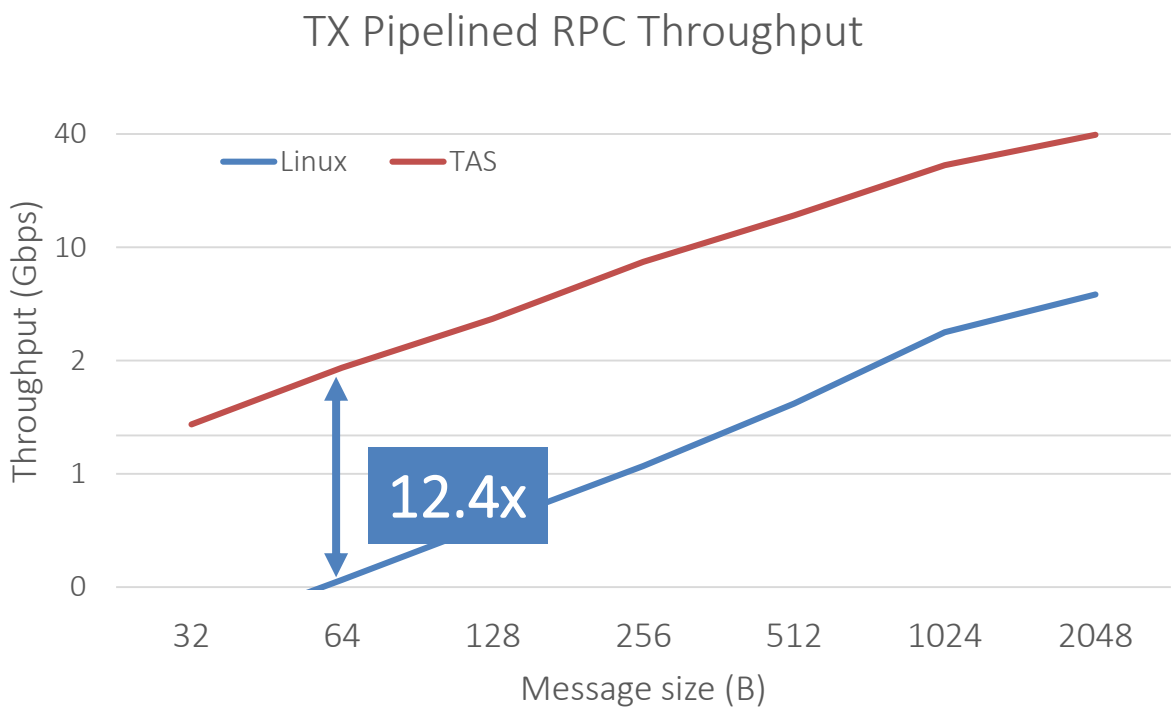




Linux vs TAS on RPCs (1 App Core)

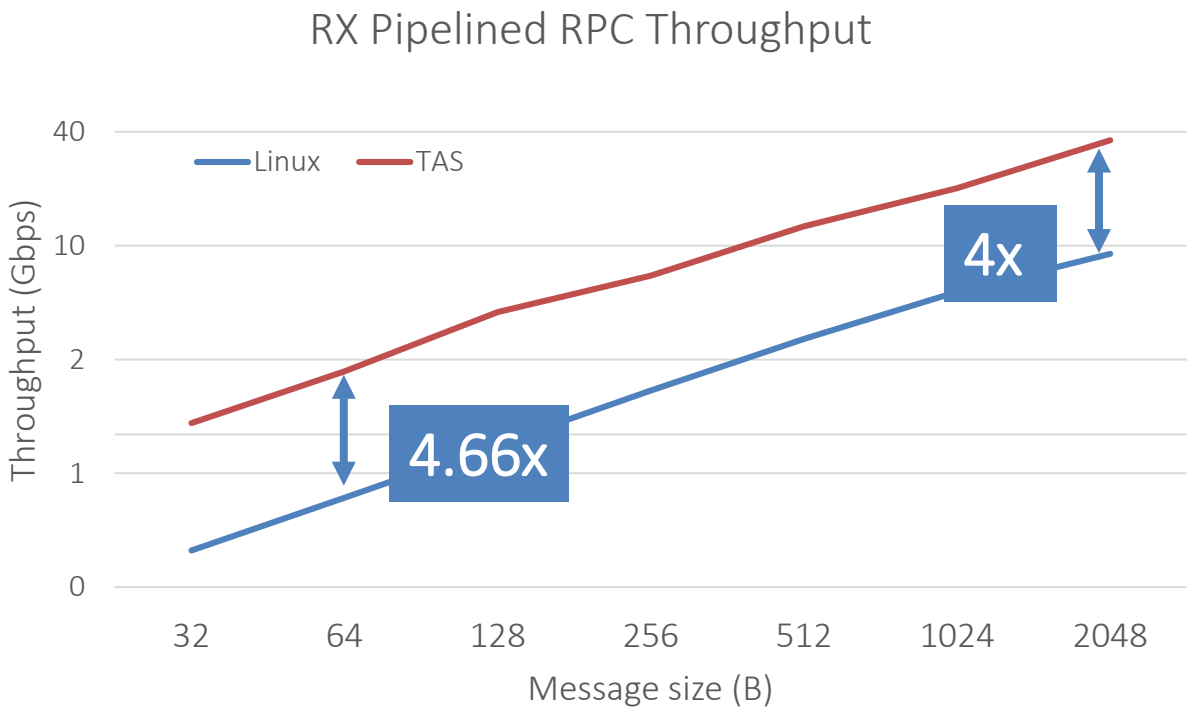
- Single direction RPC benchmark
- 32 RPCs per connection in flight
- 250 cycle application workload
- 64 bytes realistic small RPC

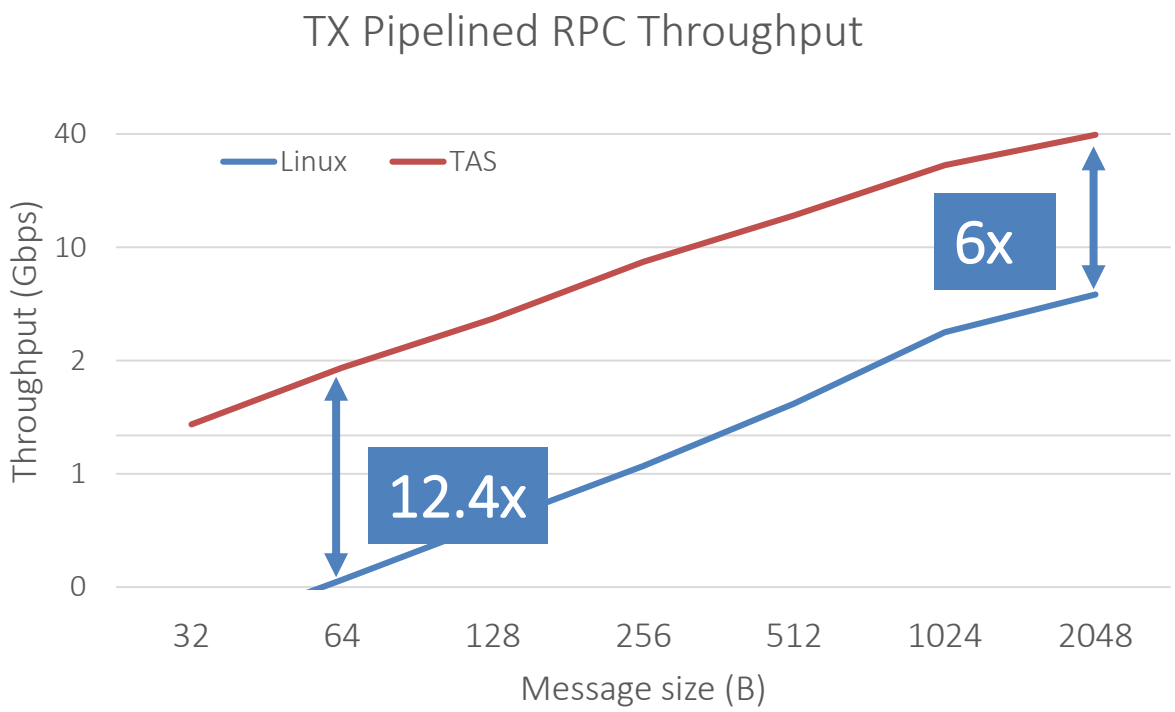




Linux vs TAS on RPCs (1 App Core)

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

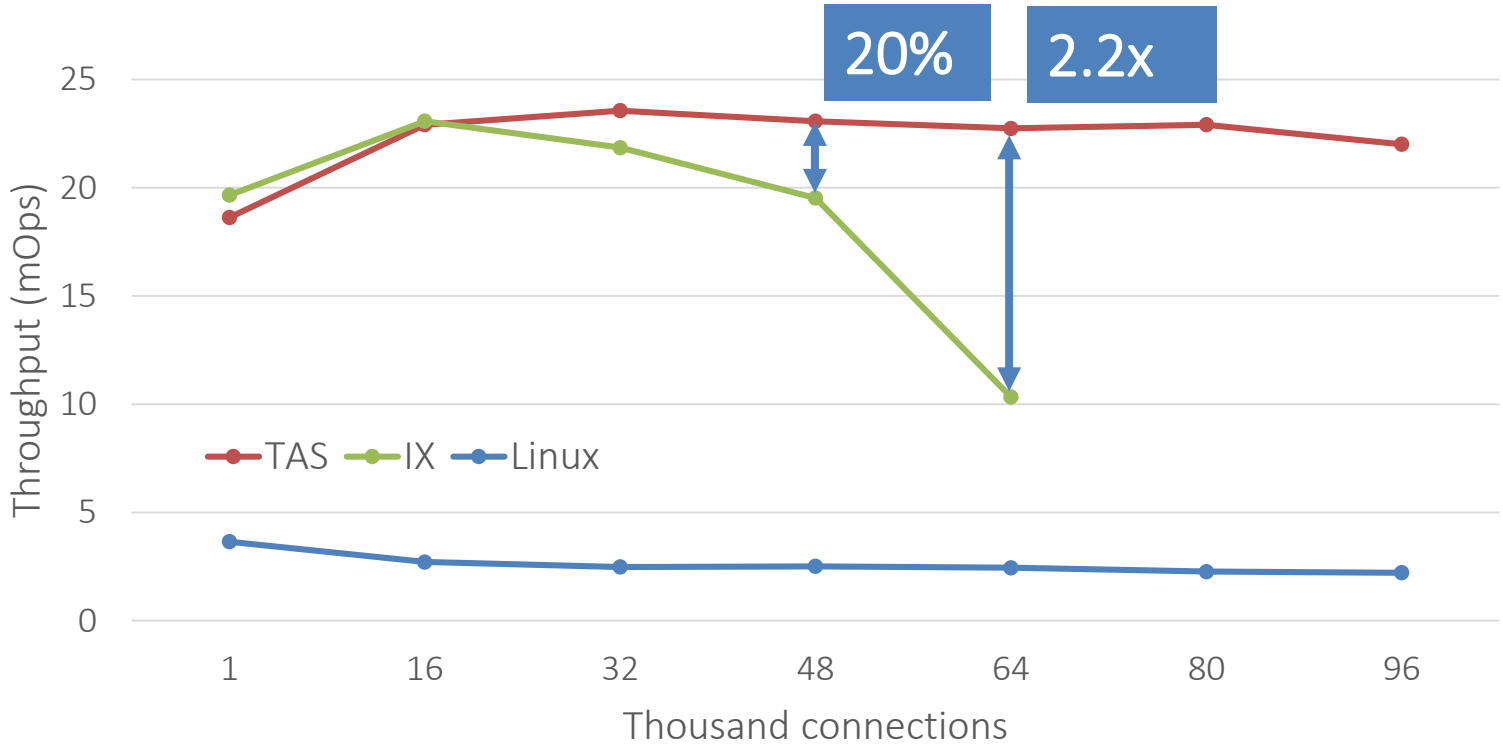




Connection Scalability

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

Key factor: minimized connection state

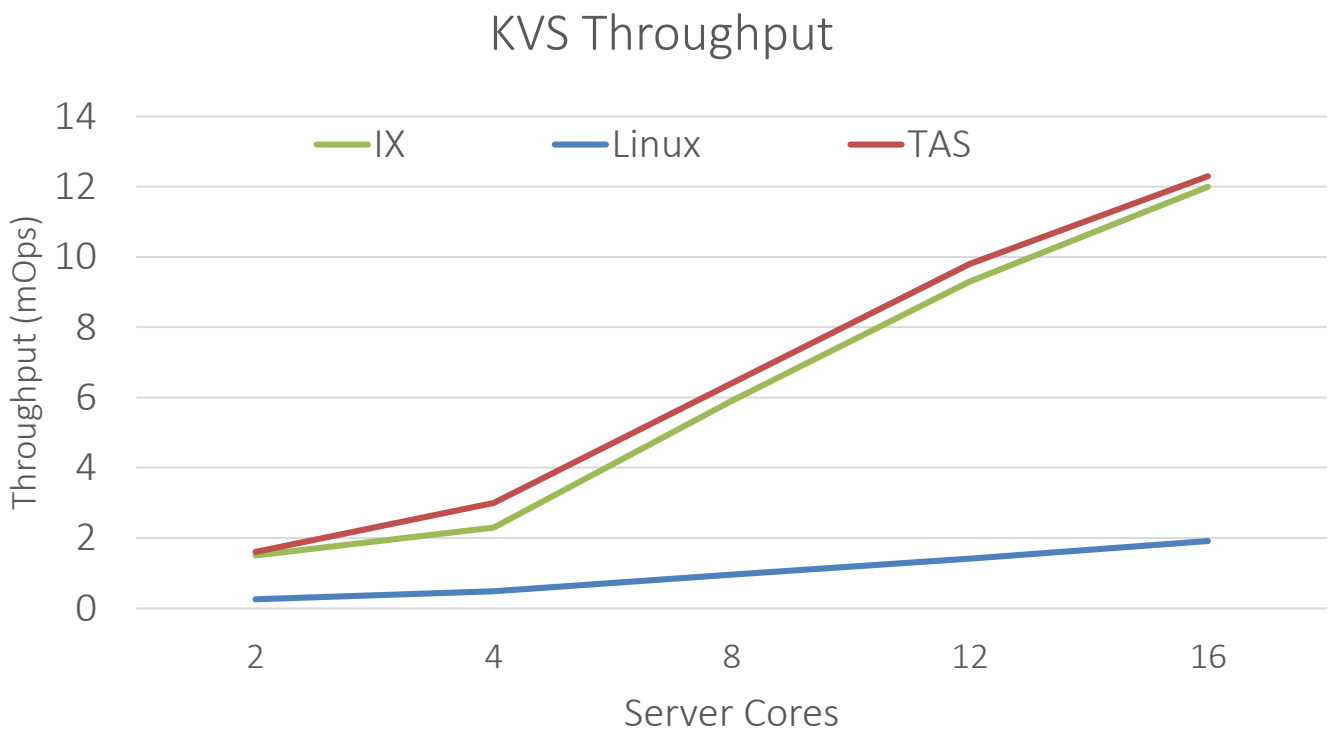


Key-value Store

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



Key-value Store

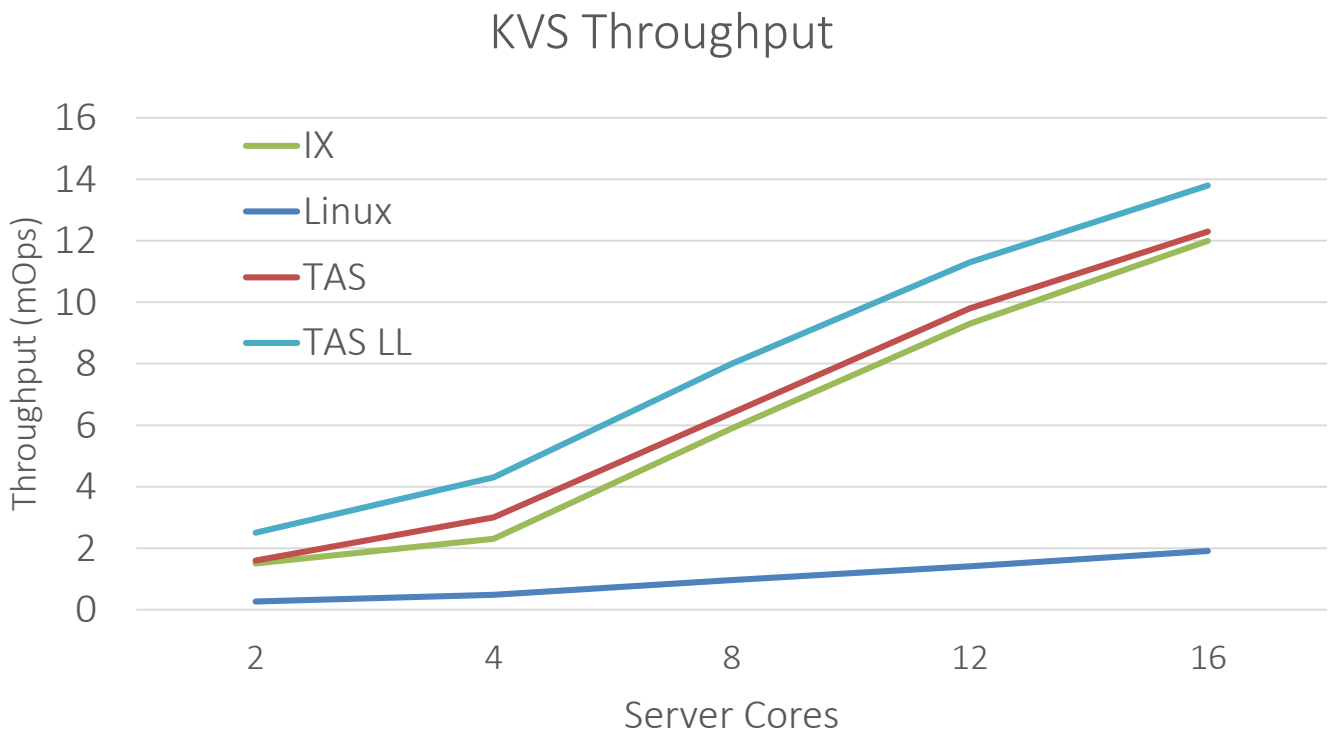
Increasing server cores with matching load (~2000 connections per core)

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

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

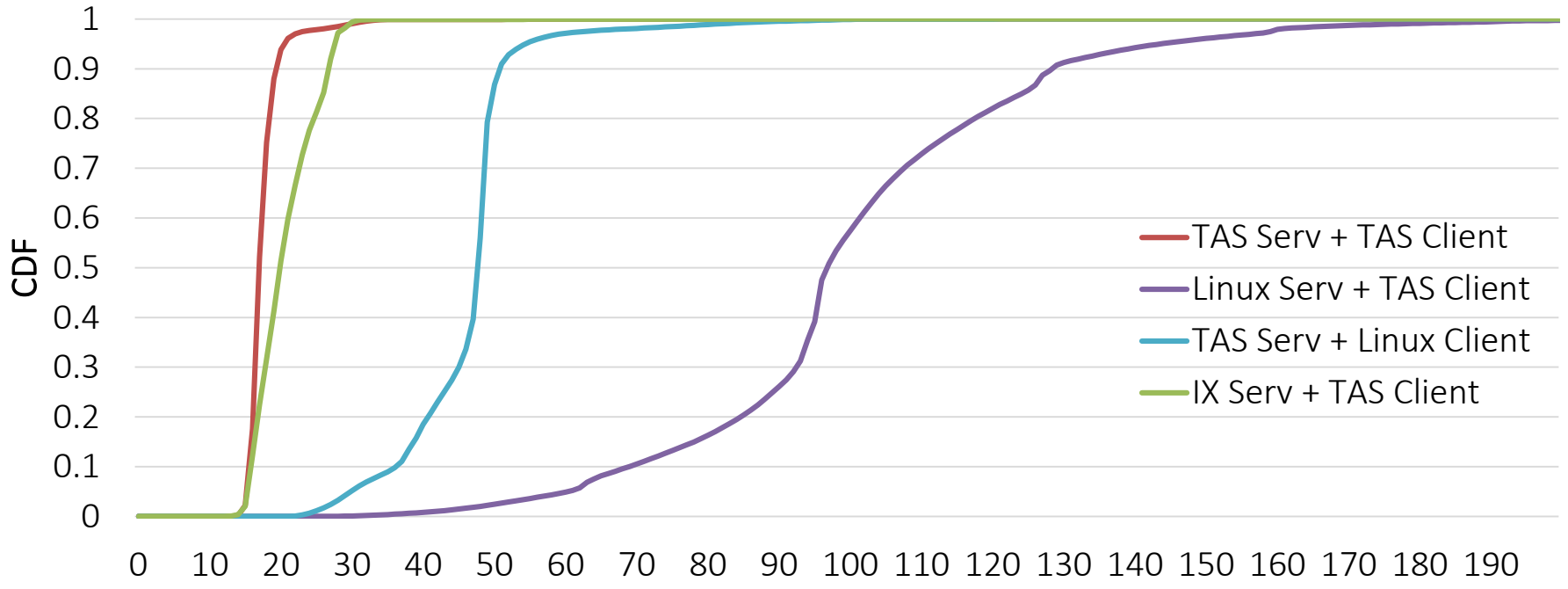
TAS has a 15-20% performance improvement over IX without sockets

- TAS: 8 app cores, 8 TAS cores



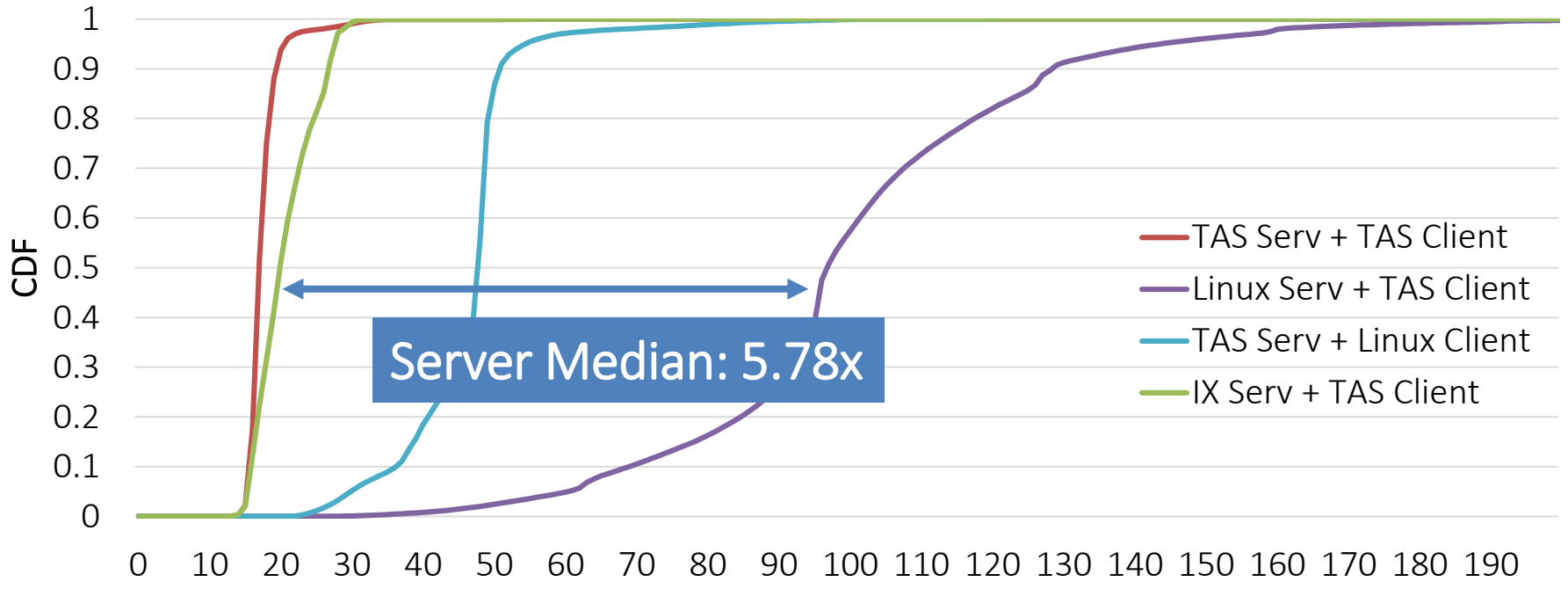
Key-value Store Latency

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



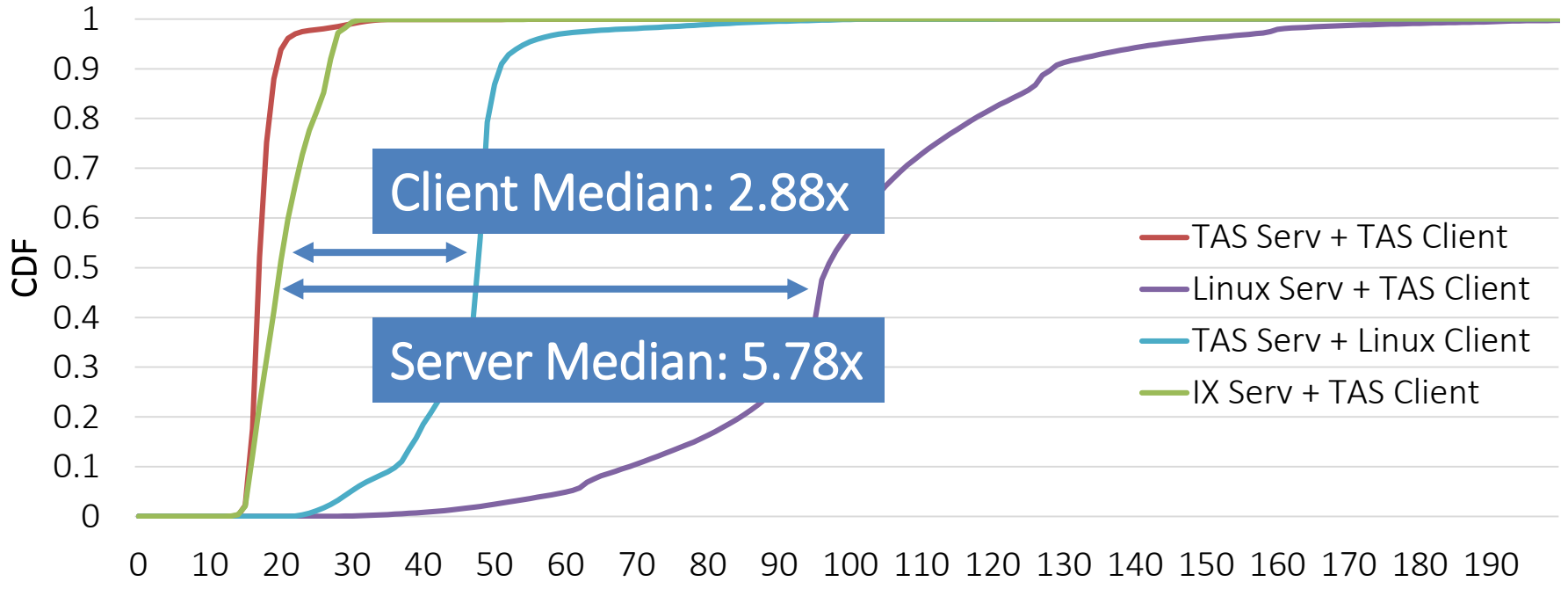
Key-value Store Latency

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



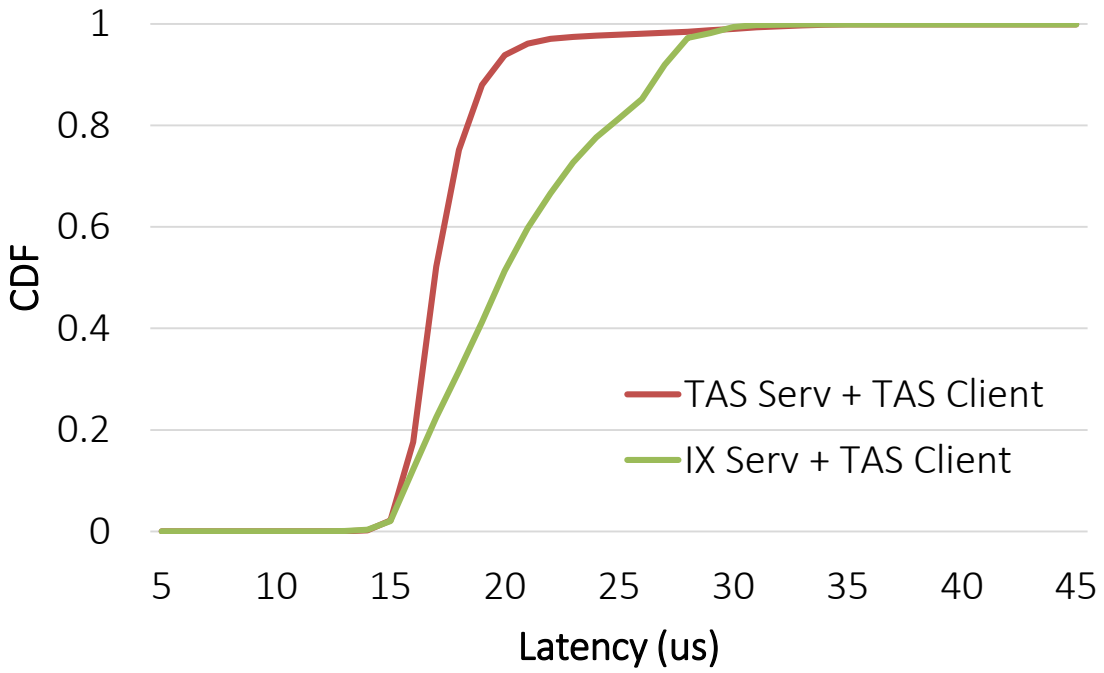
Key-value Store Latency

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



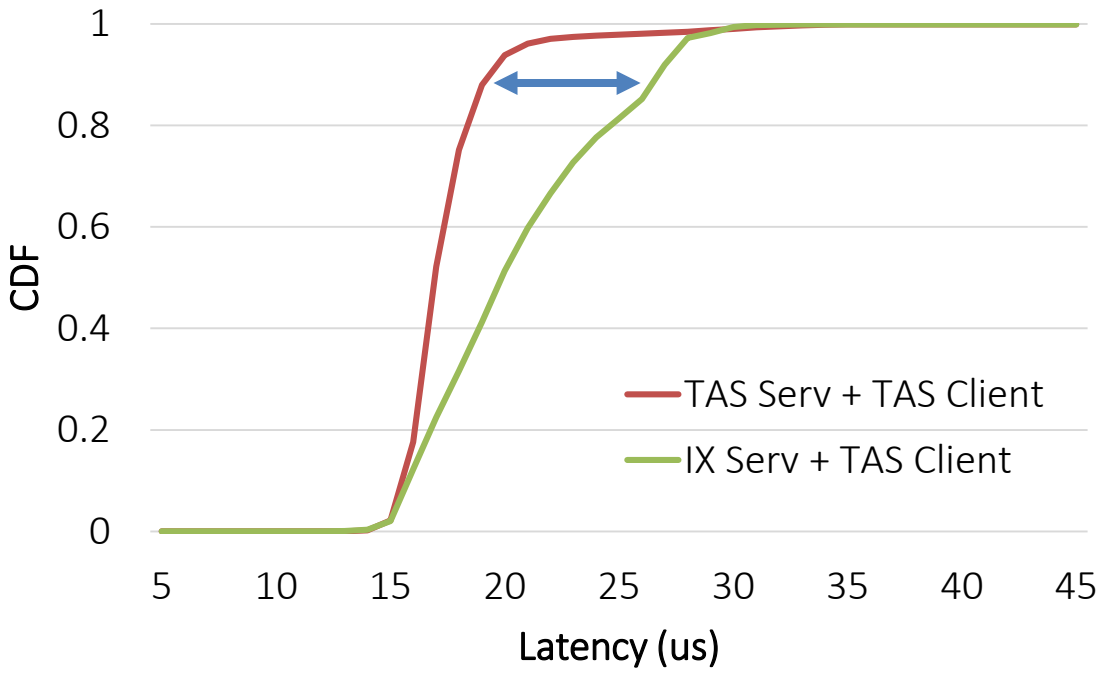
Tail Latency

IX has 50% higher latency in the 90p case



Tail Latency

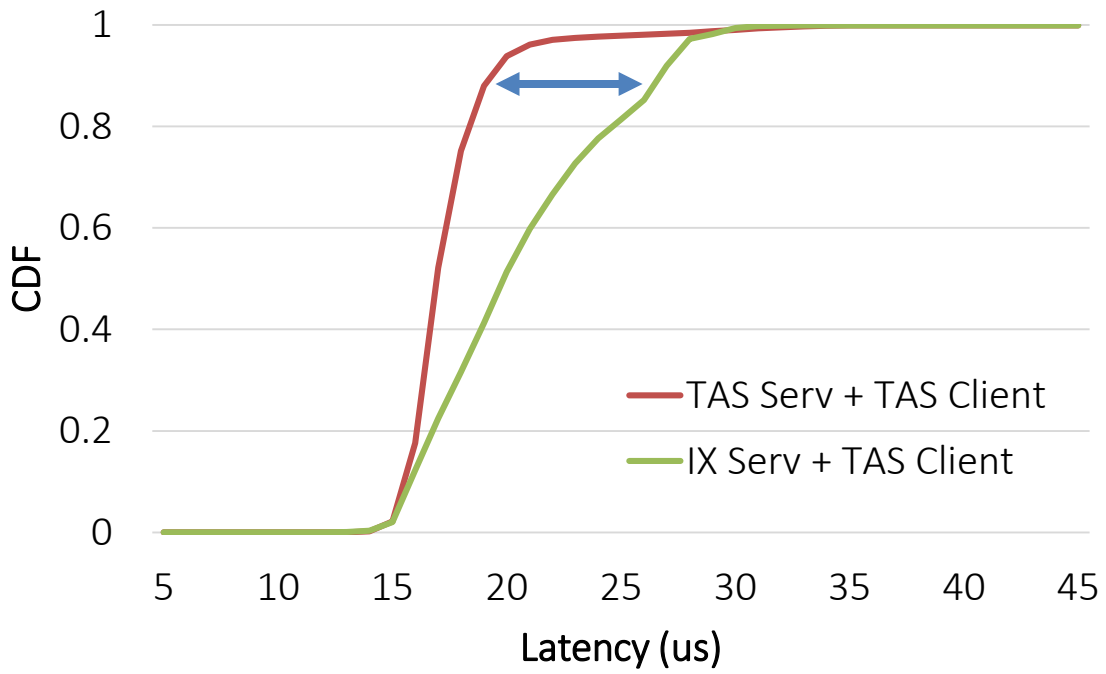
IX has 50% higher latency in the 90p case

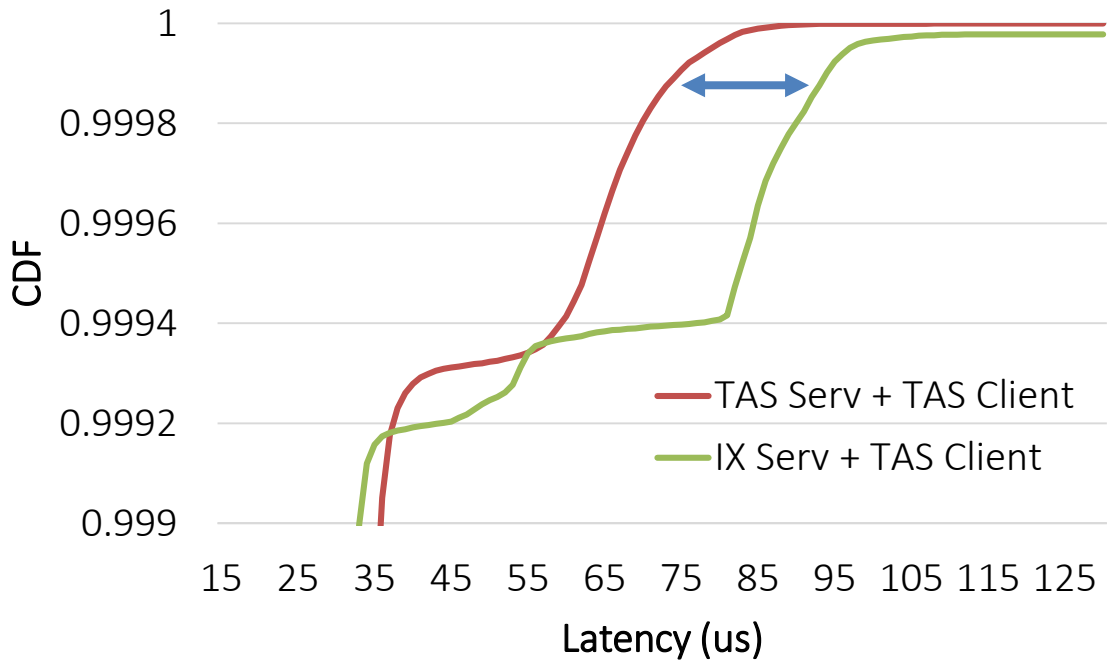


Tail Latency

IX has 50% higher latency in the 90p case

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



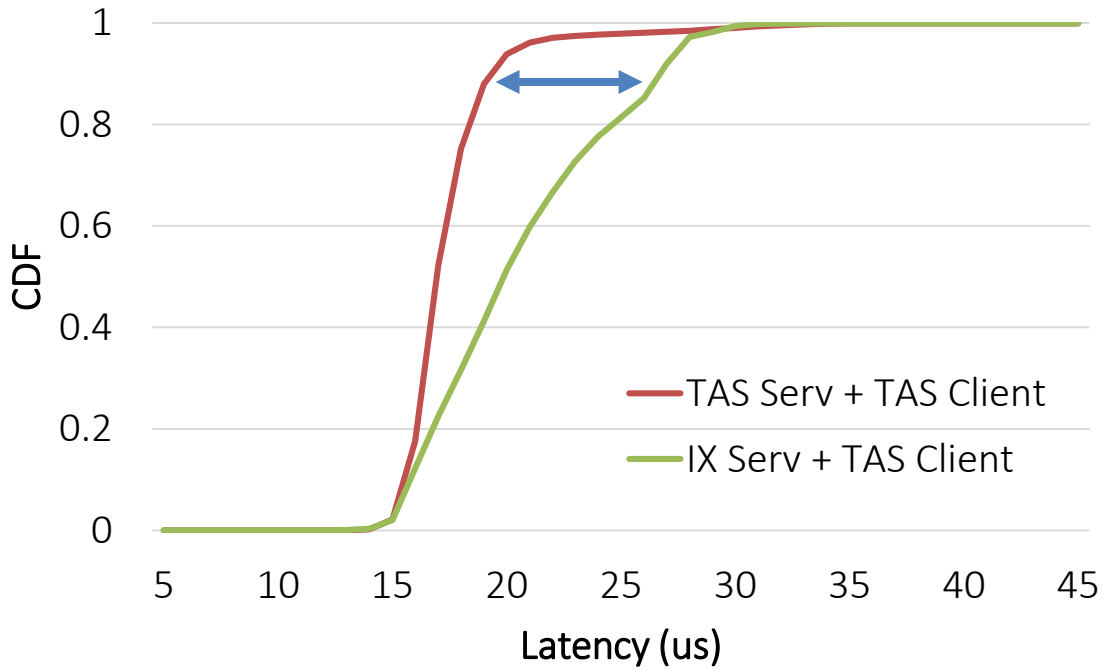


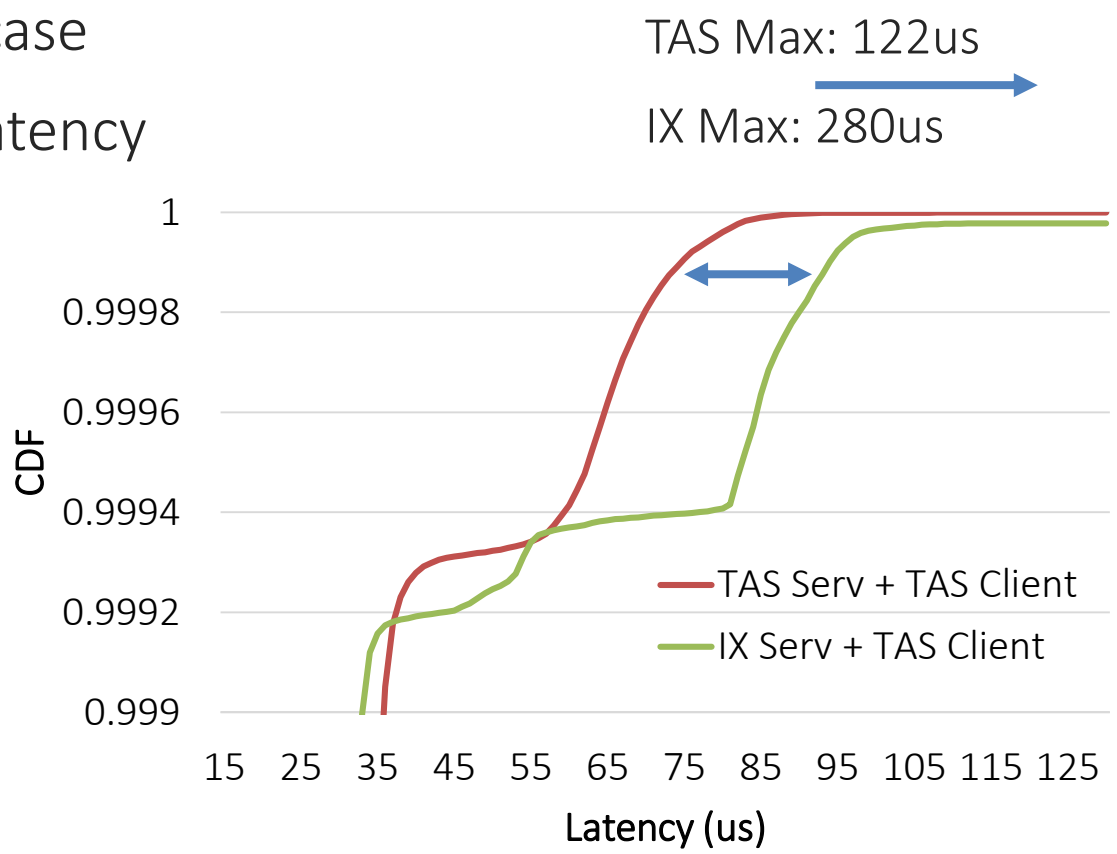
Tail Latency

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





Tail Latency

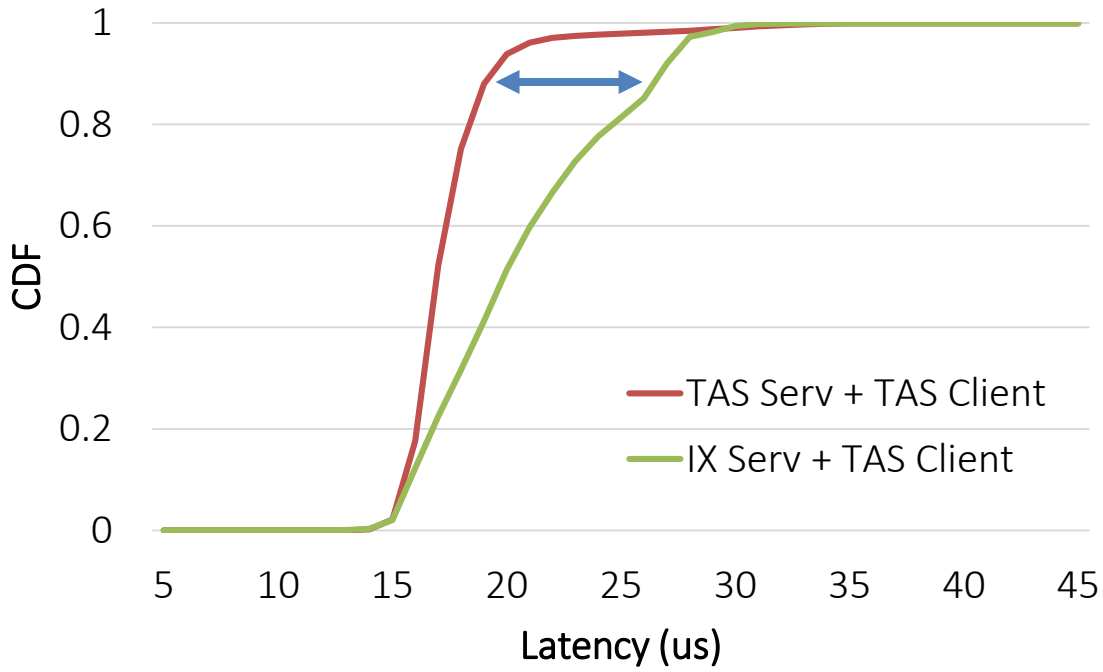
IX has 50% higher latency in the 90p case

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

In addition, IX has a 2.3x higher maximum latency

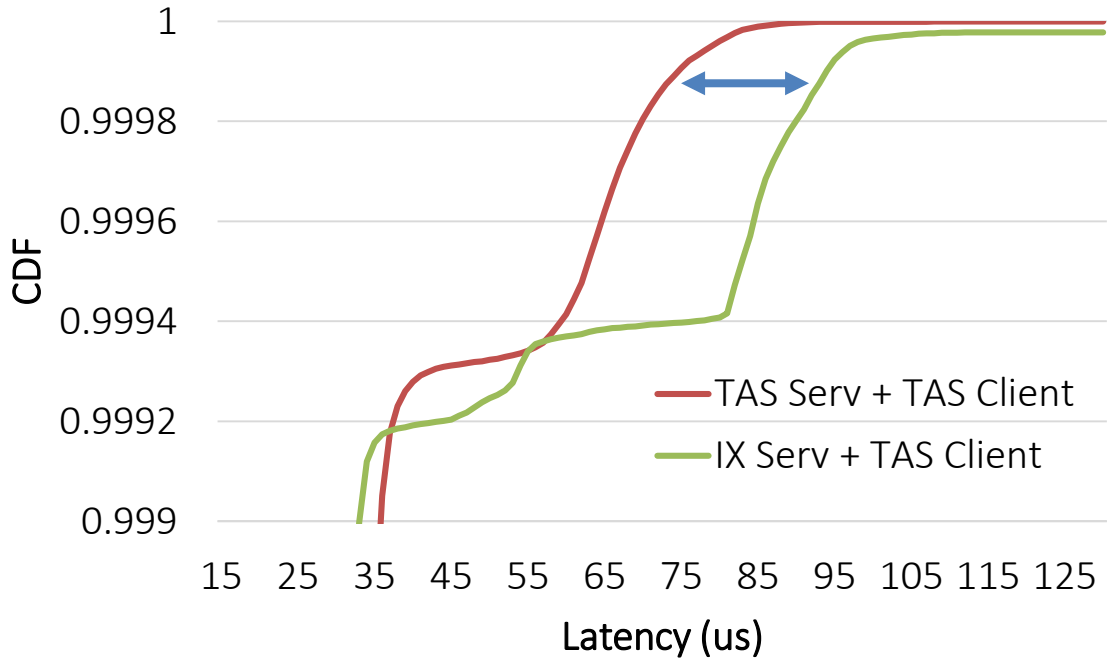
Why long IX tail?

Batching



TAS Max: 122us

IX Max: 280us

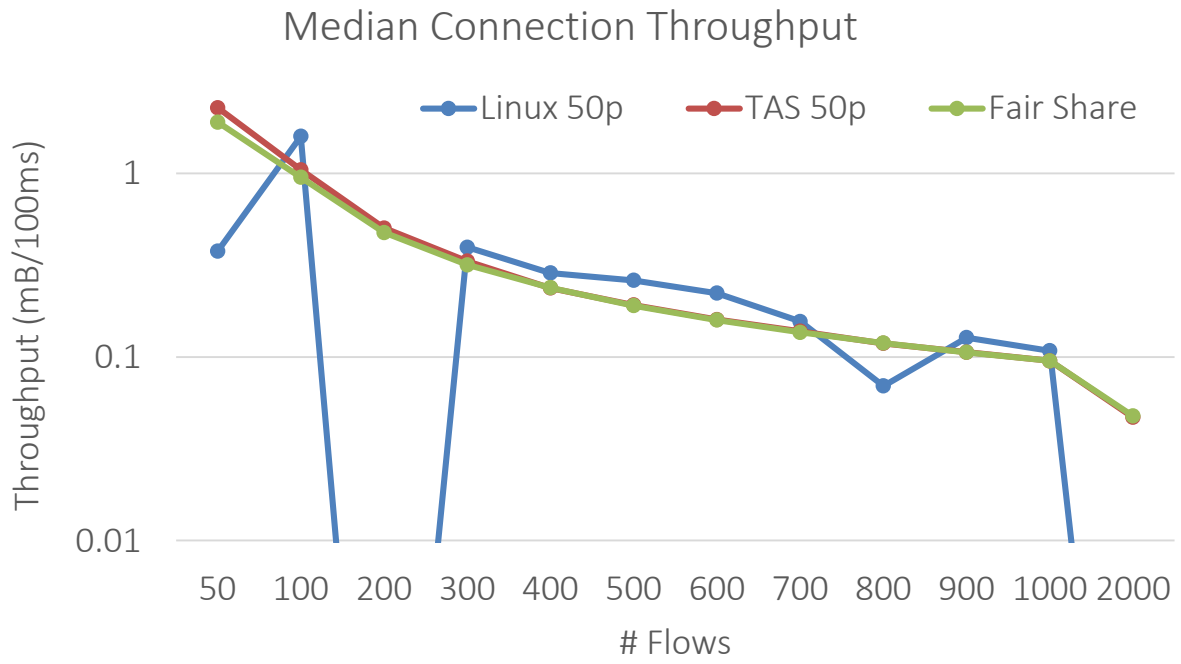


Fairness Under Incast

We want to see how TAS distributes throughput under congestion

Incast scenario, with four 10G machines all sending to one 40G server

TAS on average maintains fair throughput, while Linux is unstable

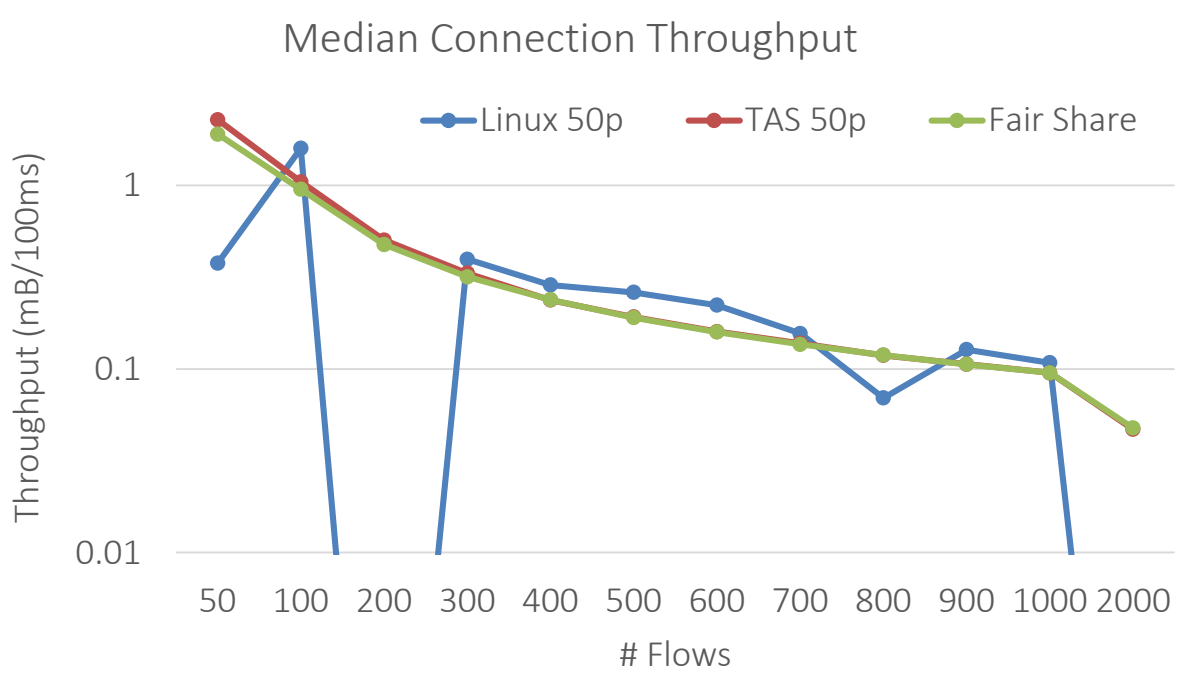


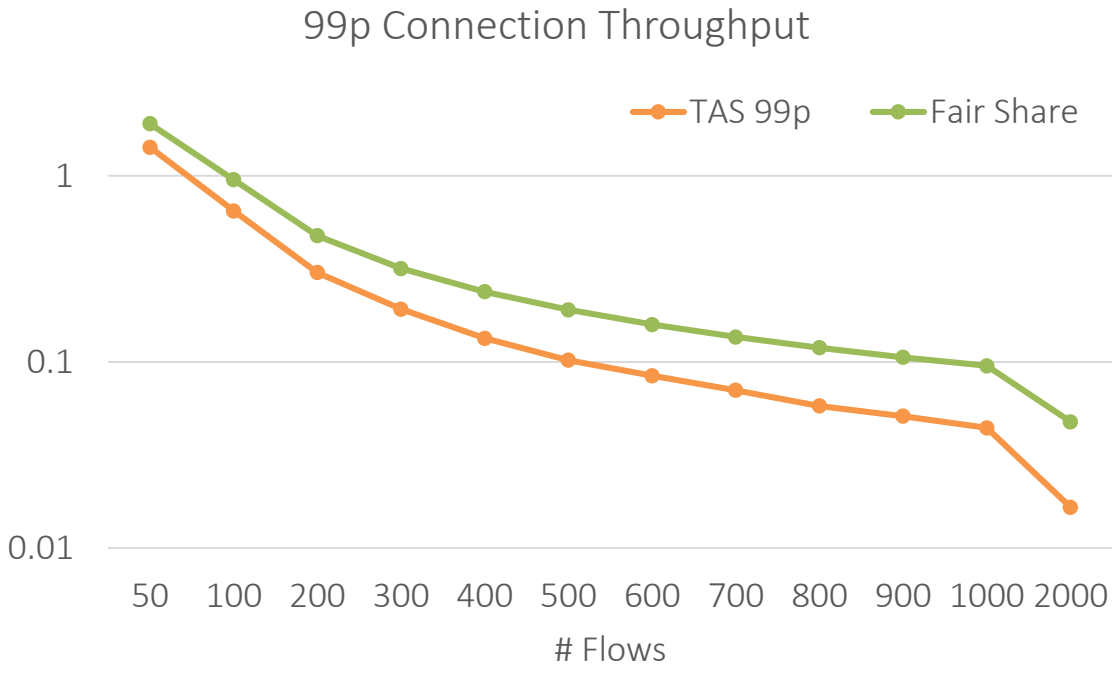
Fairness Under Incast

We want to see how TAS distributes throughput under congestion

Incast scenario, with four 10G machines all sending to one 40G server

TAS on average maintains fair throughput, while Linux is unstable



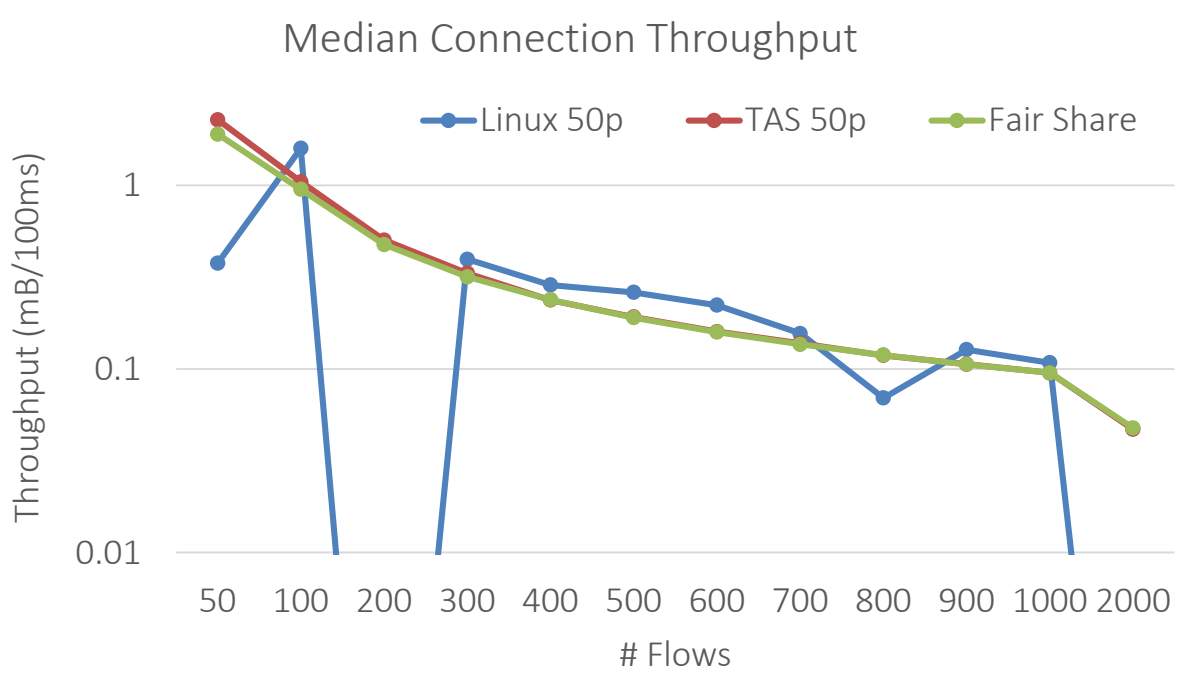


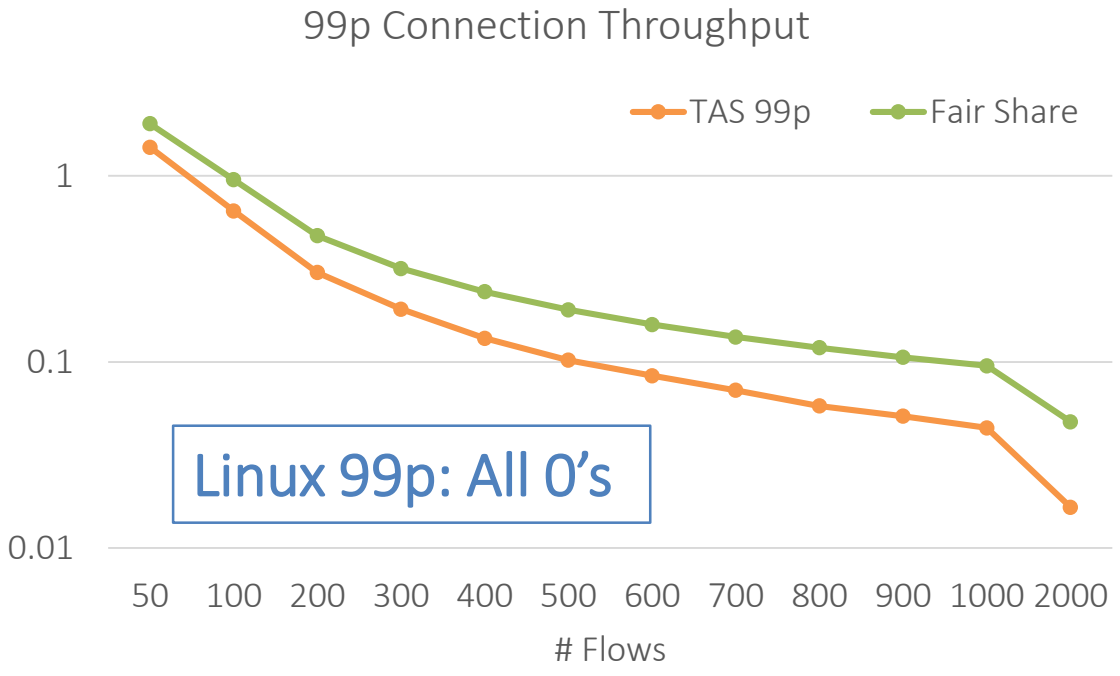
Fairness Under Incast

We want to see how TAS distributes throughput under congestion

Incast scenario, with four 10G machines all sending to one 40G server

TAS on average maintains fair throughput, while Linux is unstable





Conclusion

TAS has the convenience and features of Linux, with better performance & stability

Achieved by

1. Separating TCP packet processing into a fast and slow path
2. Minimizing connection state
3. Dedicating cores to the network stack

TAS is a purely software solution that is easy to deploy and operate

Try it yourself!

https://github.com/tcp-acceleration-service