The Future of Cloud Networking is Systems

Dan Ports
Microsoft Research

https://drkp.net/

I am a distributed systems researcher.

This is a systems conference.

…so why am I giving a talk about networking?

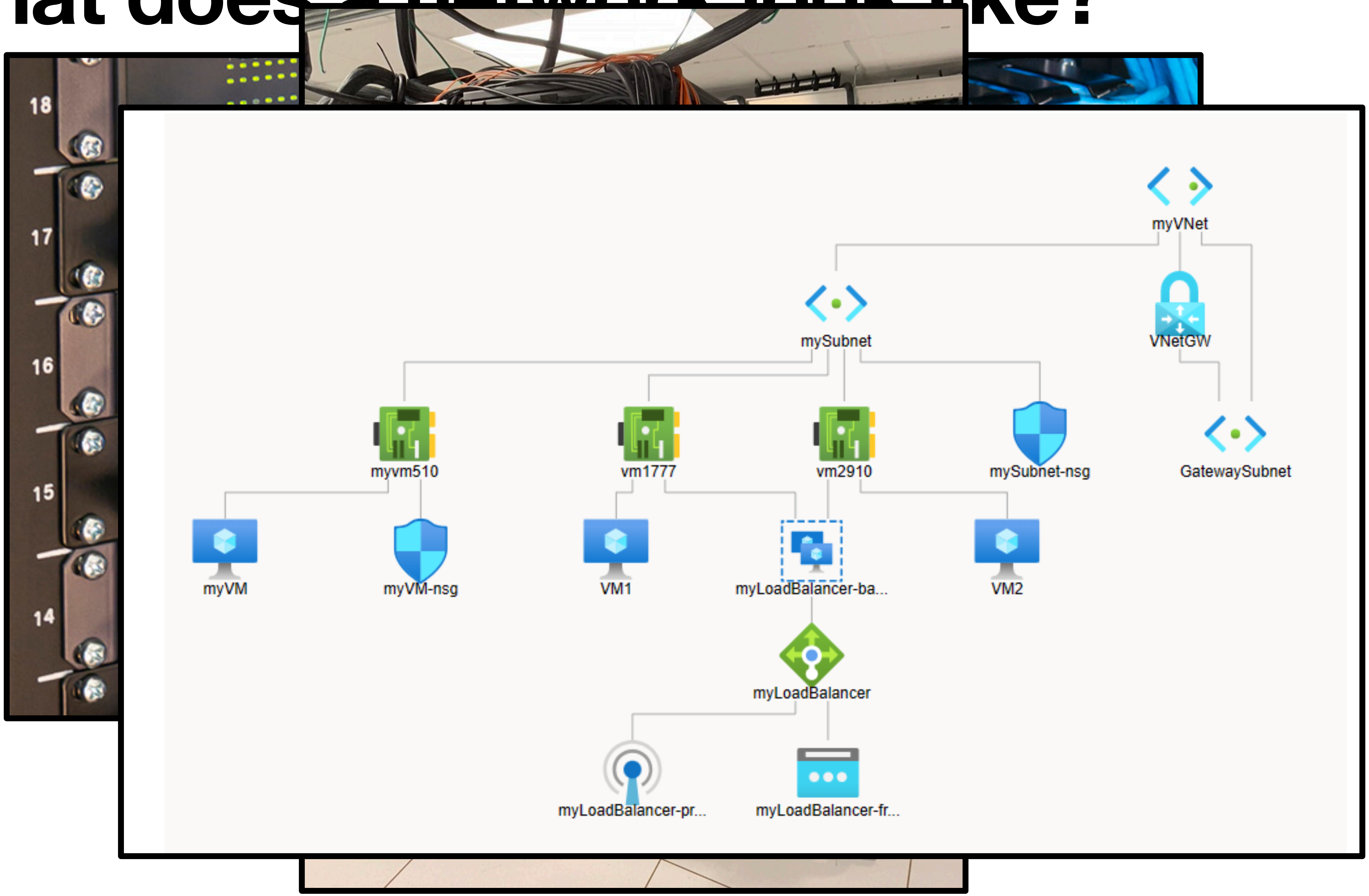
Systems and networking research have **converged**

- Cloud networks rely on huge distributed systems
- Networks can offer new features for distributed systems

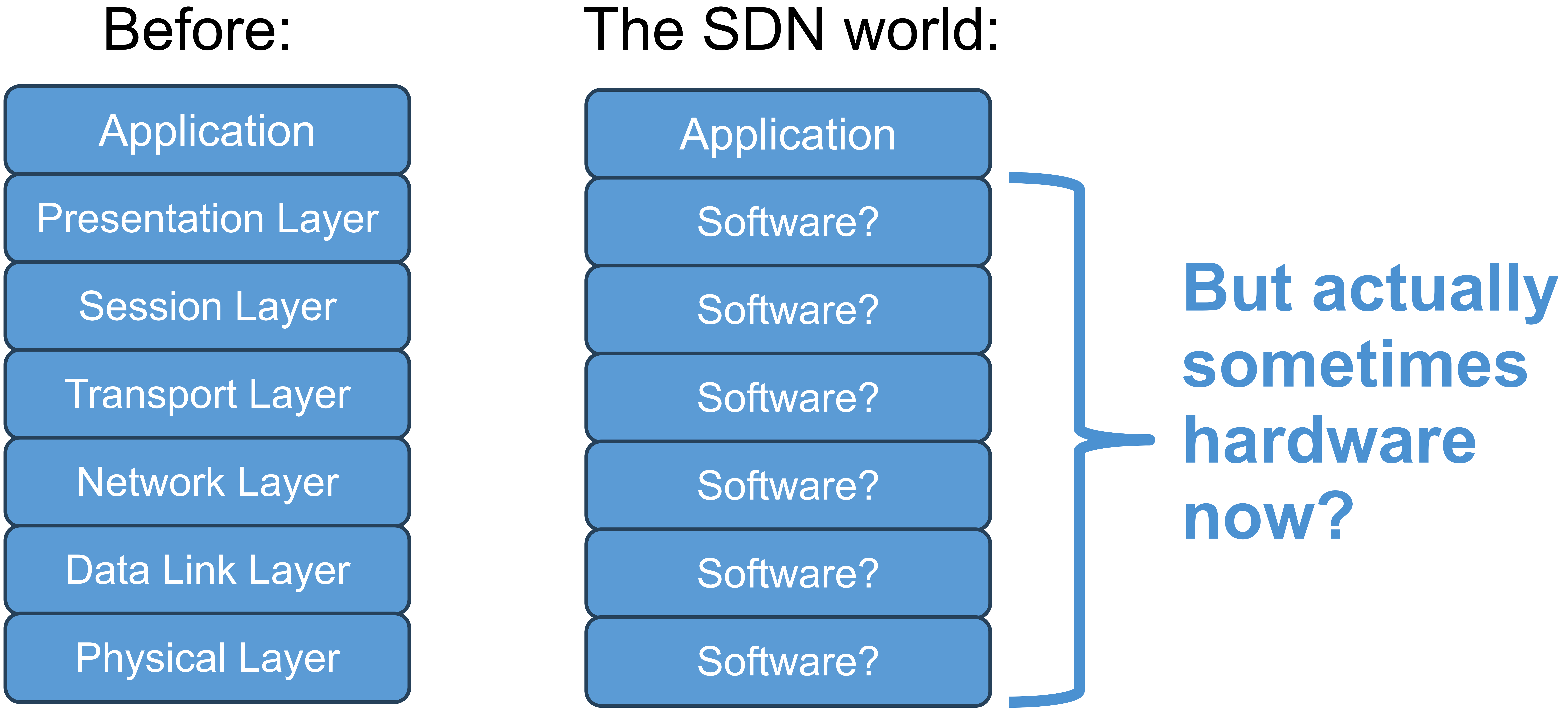
Exciting possibilities for research at this intersection

Hyperscale datacenters have changed the computing landscape

What does a network look like?



The modern network stack is fully abstract



Layers of virtualization in a modern cloud network

Physical fabric: a highly multi-path L3 routed network

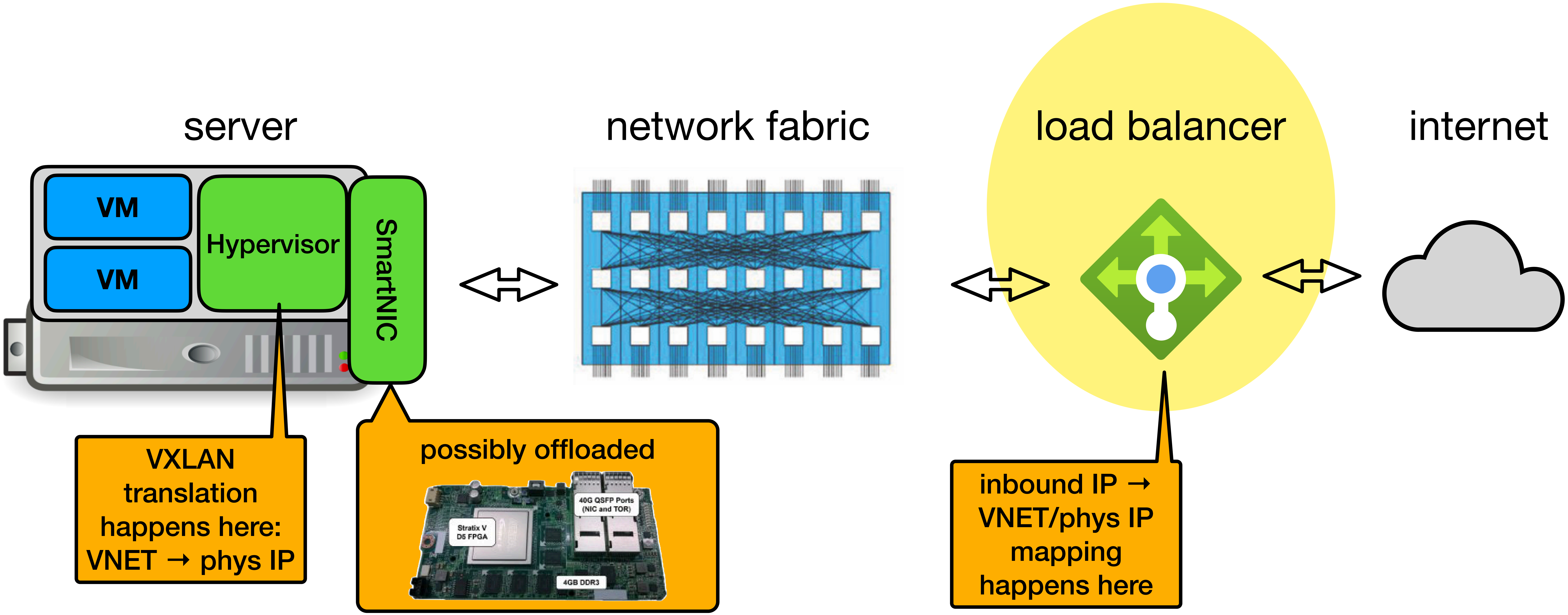
Network virtualization (VXLAN): isolate tenants and hide physical topology

- internal customer VNET IP → physical datacenter IP

Load balancers and NAT: provide external access to networked resources

- public IP address → one or more IPs on a customer's VNET

(Partial) anatomy of a datacenter network



Load balancers are central to cloud networks

They are the gateway to most deployed cloud services.

They process most inbound traffic to the datacenter.
(Not just classic load balancing — other network functions like NAT and DDoS too)

They are inherently disaggregated (not tied to a single server)

…and, of course…

Load balancing strategies and algorithms have always been a fundamental problem in building high-performance distributed systems

…which means that…

Building a cloud-scale load balancer is both a major efficiency challenge *and* an opportunity to unlock powerful new functionality for distributed systems!

Evolution of cloud networking infrastructure

off-the-shelf solutions: small-scale, expensive
(e.g. load balancer boxes)

cloud-scale software implementations

two conflicting pressures

advanced new features

increased efficiency

Programmable hardware can help us meet both requirements!

Classic *software* load balancer design

5-tuple → action

e.g.,

67.40.19.211:1024,20.83.140.166:80 ->
[encap, VNET#42, 10.0.0.1]

Data plane
(servers)

executes flow
transformations

Flow
transformations

Control plane

(decides what
action to take
with a flow)

see, e.g. "Ananta: Cloud-Scale Load Balancing" [Patel et al., SIGCOMM '13]

New programmable network hardware can help

Smart NICs / DPUs
(Mellanox BlueField, AMD/Pensando Elba, Intel IPU, …)
~400 Gbit/s per device

Programmable switches
(Intel Tofino, Mellanox Spectrum, Cisco Silicon One, …)
~10-50 Tbit/s per device, limited memory

Combinations of these devices are possible too

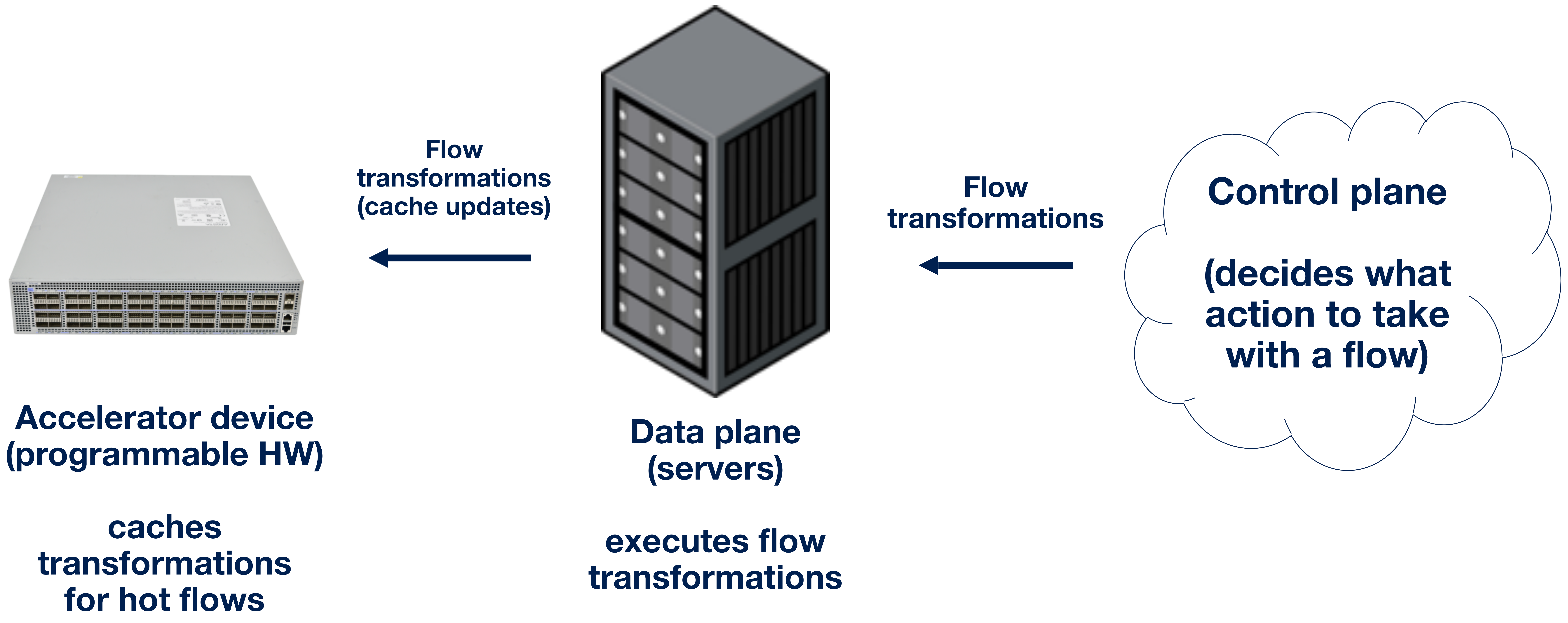
New programmable network hardware can help

Commonalities between architectures

- Optimized packet processing accelerator runs simple “programs” at line rate
- **Flexible beyond “traditional” network processing, e.g. IP routing**
- **Can make dynamic, per-packet decisions**
- Packets that can’t be processed in hardware can be sent to onboard CPU cores or external systems

Smart NICs can have access to greater memory and onboard CPUs;
programmable switches have higher packet processing rates but limited resources

Accelerated load balancing architecture



Research challenges for accelerated load balancing

How do we make it work ***efficiently***?

- Which flows do we cache?
  Accelerator HW can handle many packets, but limited flow state

ML-based flow classification to trigger offloads

How do we make it work ***correctly***?

- How do we ensure consistent states between accelerator and SW?

Distributed cache consistency protocol for managing flow state

How do we make it work ***flexibly***?

- Can we support multiple HW platforms with different properties

Platform-independent specification of desired packet transformation behavior

What new things can we do with a fast, flexible load balancer?

Opportunities

~~Research challenges~~ for accelerated load balancing

We can build new load balancing policies
customized for applications

We can run flexible load balancing at microsecond scale
using new hardware accelerators

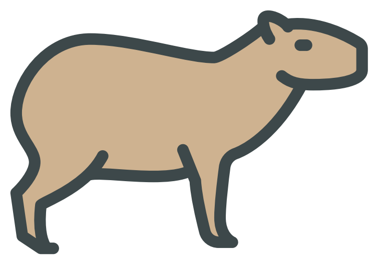
**We can use these to make distributed systems
faster, more efficient, and more reliable!**

Agenda for this talk

Overview of accelerated load balancing

Three systems that enable new functionality with accelerated load balancing

Pegasus: balancing skewed workloads in distributed storage



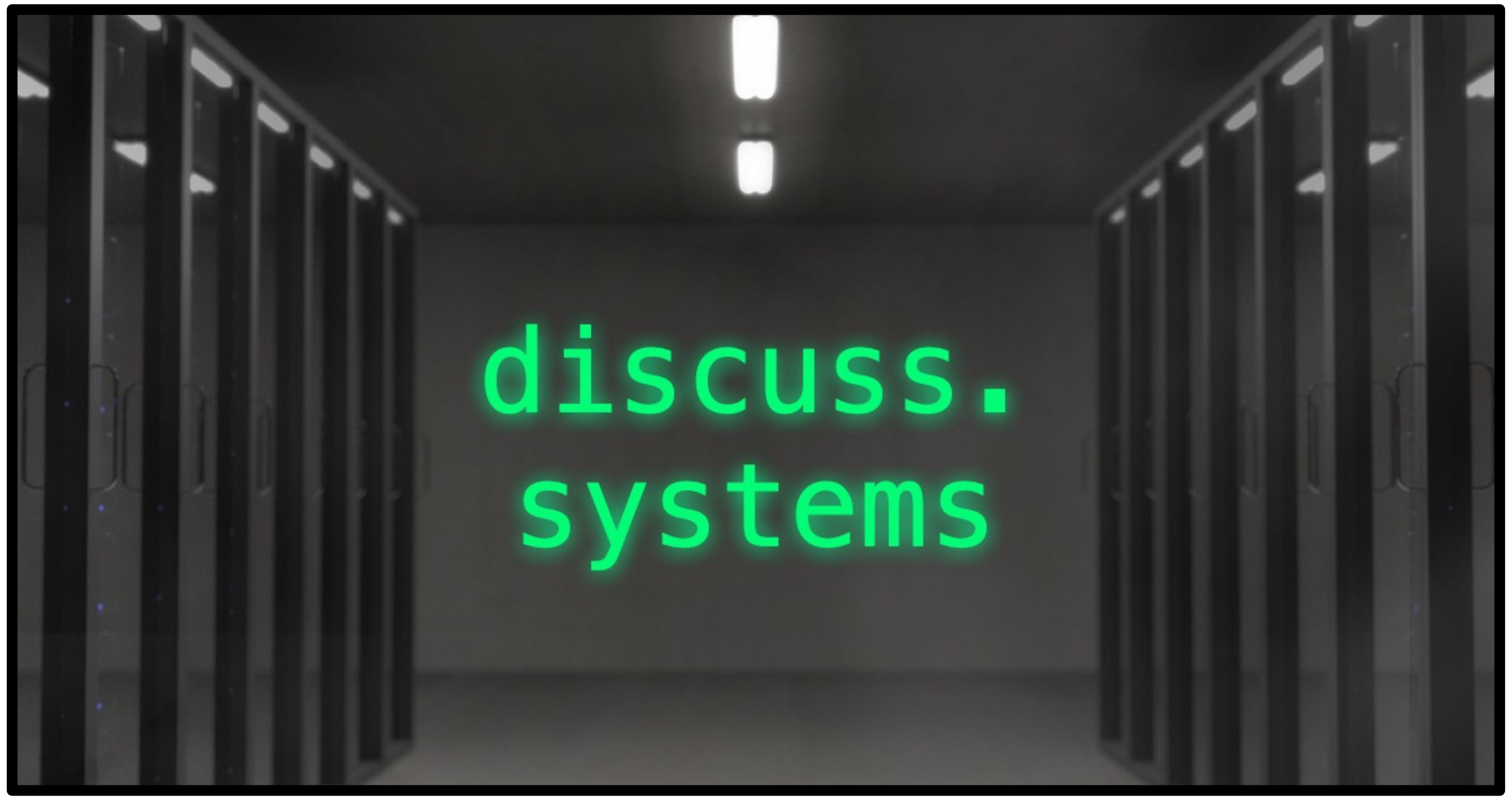
Capybara: live migration of active TCP connections at µs-scale

Beaver: using load balancers to take practical persistent checkpoints

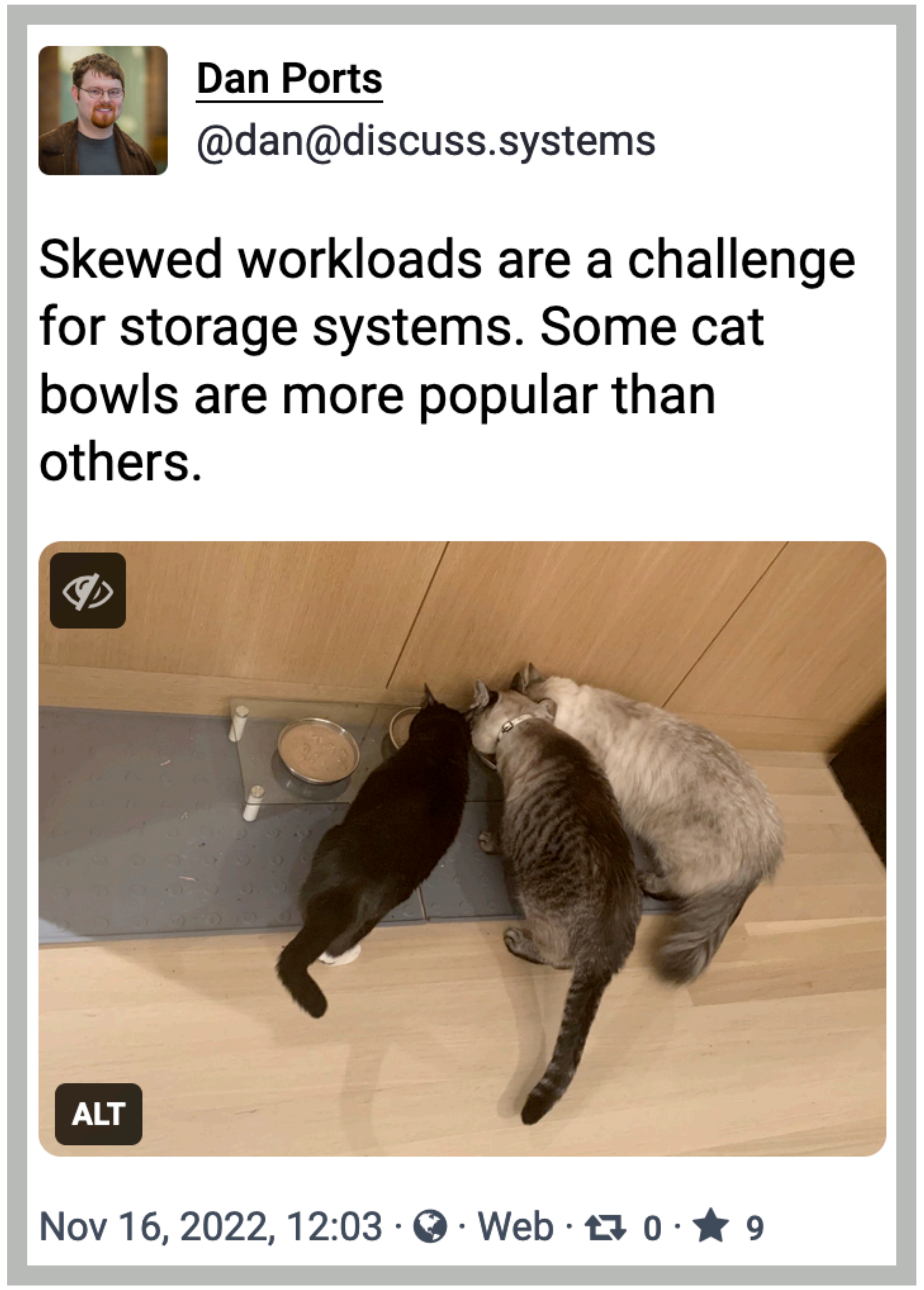
My other job

In my spare time, I run a social network for systems researchers

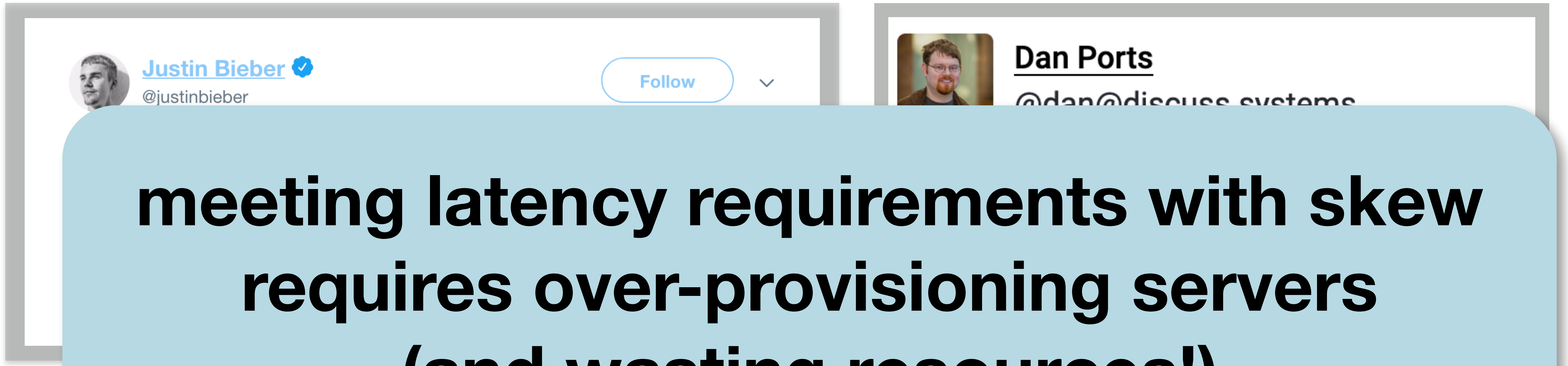
(You should join! - https://discuss.systems/)



Many workloads are skewed and dynamic



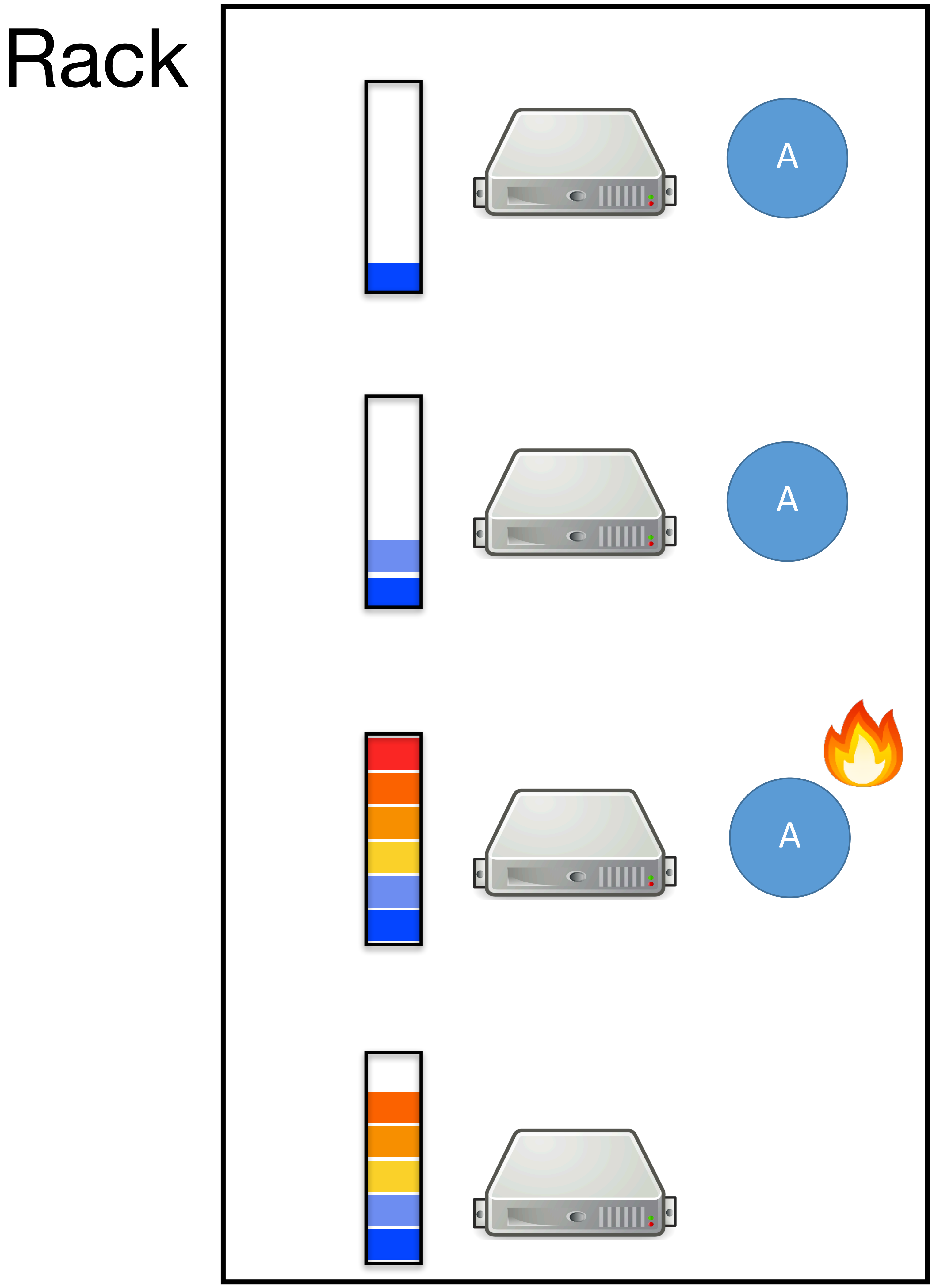
Skewed workloads lead to load imbalance



meeting latency requirements with skew requires over-provisioning servers (and wasting resources!)

rack-scale
storage system

Observation: rack as a whole has spare processing capacity



How to route requests to the right server?

How to ensure consistency?

Our approach: Pegasus

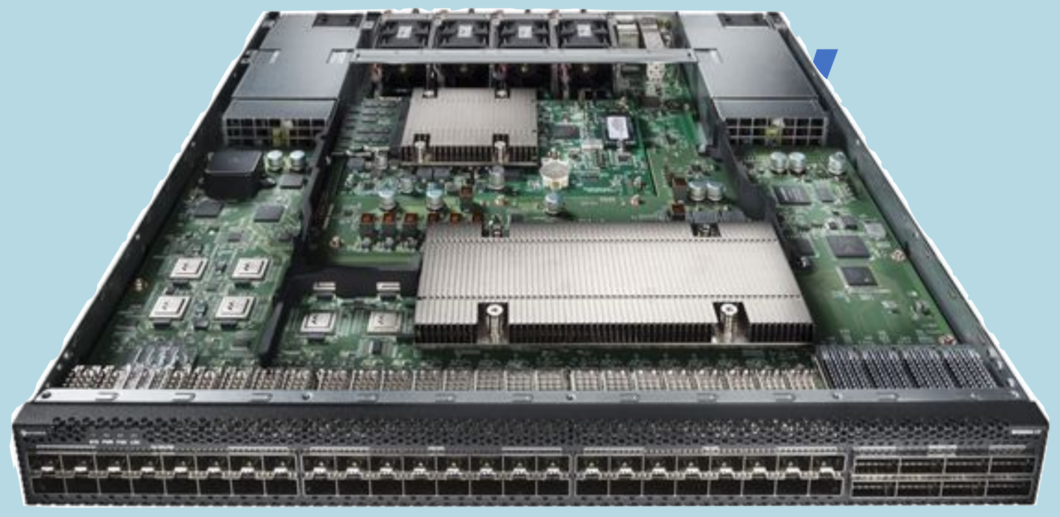
rack-scale
storage system

programmable
top-of-rack switch
as
load balancer

selective replication

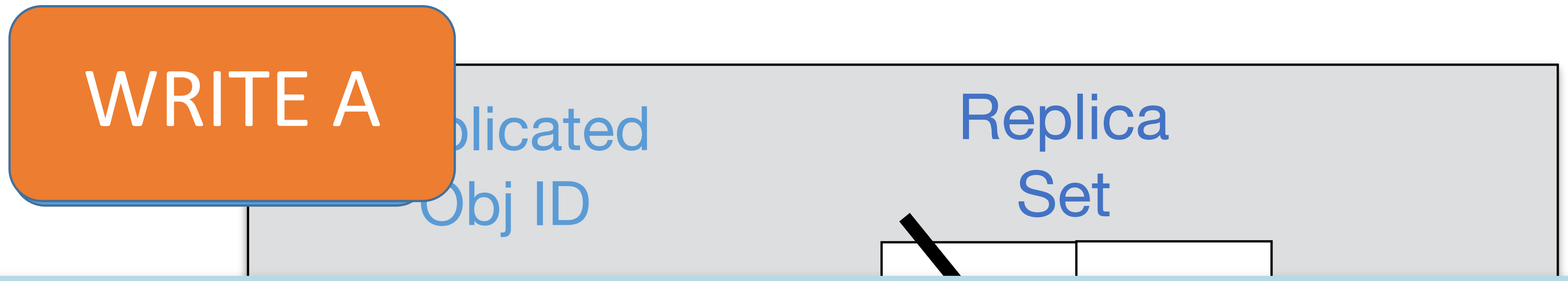
via

in-network coherence



[J. Li et al, "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories, OSDI'20]

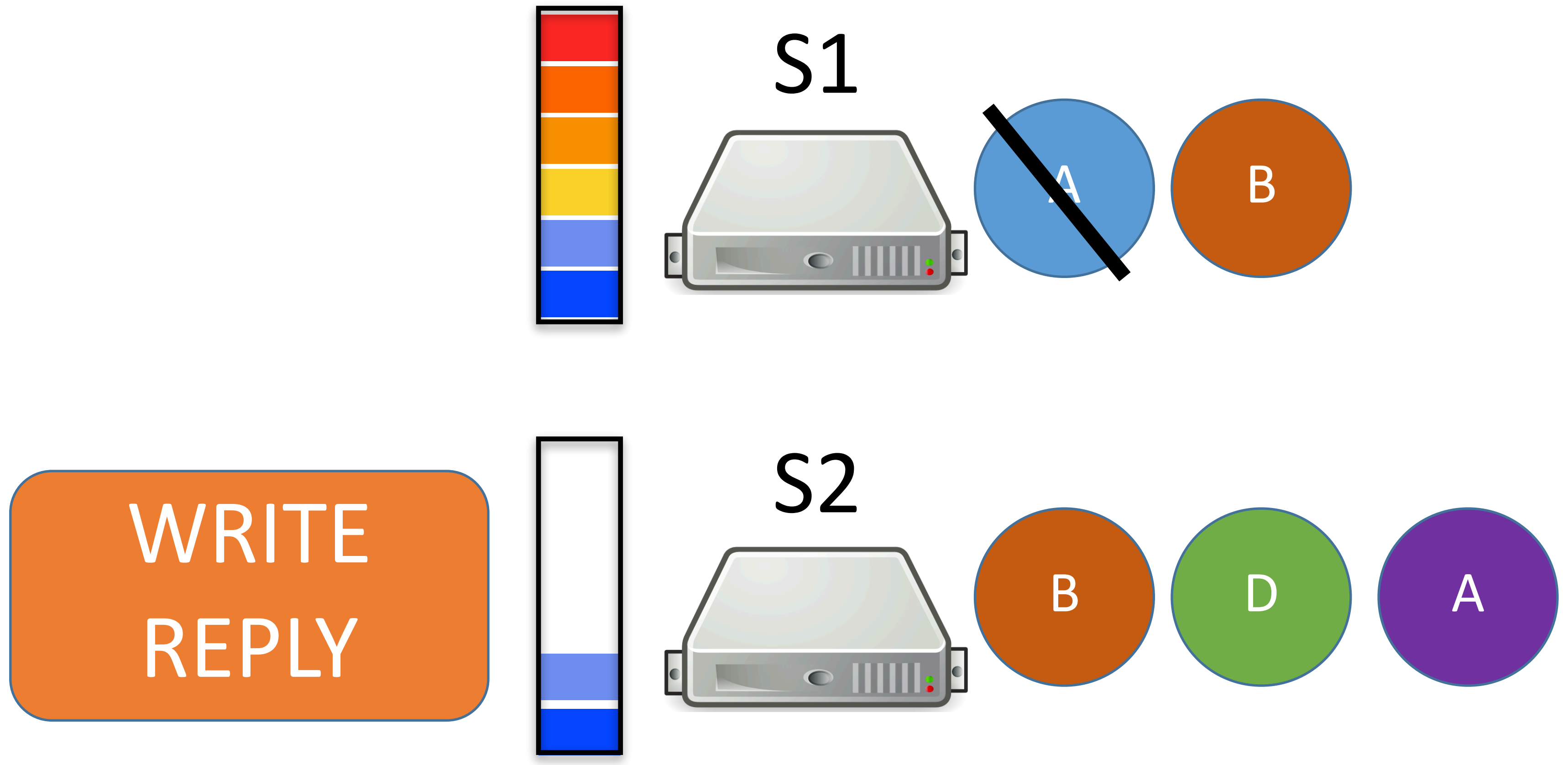
Coherence Directory Approach



Challenges:

Where to implement the coherence directory?

How to design an efficient coherence protocol?



Coherence Directory Approach

We can put an object anywhere, as long as we keep track of where we put it

We can make as many copies as we want, as long as we keep track of where they are

We can move an object as frequently as ***on every put operation***

In-Network Coherence Directories

rack-scale
storage system

- All requests and replies traverse the ToR switch
- ToR serves as a **central point**
- **Line-rate** packet processing
  - No throughput bottleneck
  - Zero latency overhead

Pegasus Coherence Protocol

Load balancer processes *all* requests

LB maintains coherence
keeping track of which r
(using version numbers)

Requests are forwarded

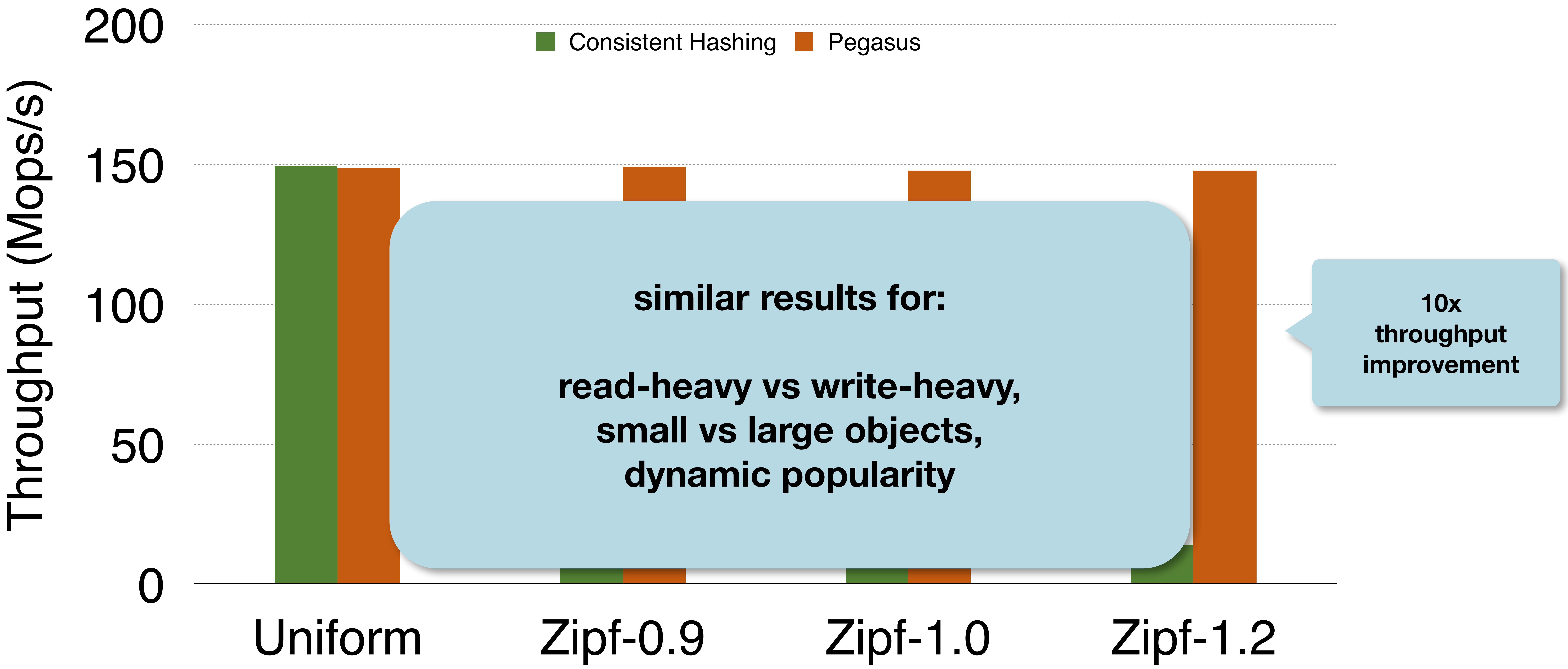
- read requests: forwa
- write requests: pick a
  and update directory
  then update the directory once complete

Protocol benefits:

- **Guarantees linearizability**
- **One RTT**
- **Non-blocking**
- **No extra coherence traffic**

Pegasus Balances Highly Skewed Workloads

28-server KV store, YCSB read-only workload, 50 μs latency SLO



Pegasus Summary

Specialized load balancer application for highly skewed workloads

Pegasus leverages the central vantage point of the network switch
to keep track of where data is located and which servers have capacity

Enables a new, co-designed coherence protocol

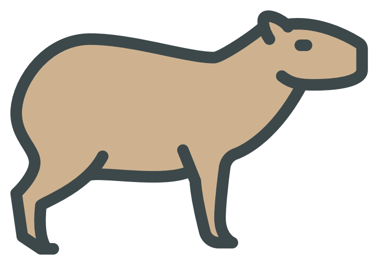
Result: a system that can handle **skewed workloads**
with the performance of a **uniform workload**

Agenda for this talk

Overview of accelerated load balancing

Three systems that enable new functionality with accelerated load balancing

Pegasus: balancing skewed workloads in distributed storage



Capybara: live migration of active TCP connections at μs-scale

Beaver: using load balancers to take practical persistent checkpoints

A challenge for load balancing: TCP

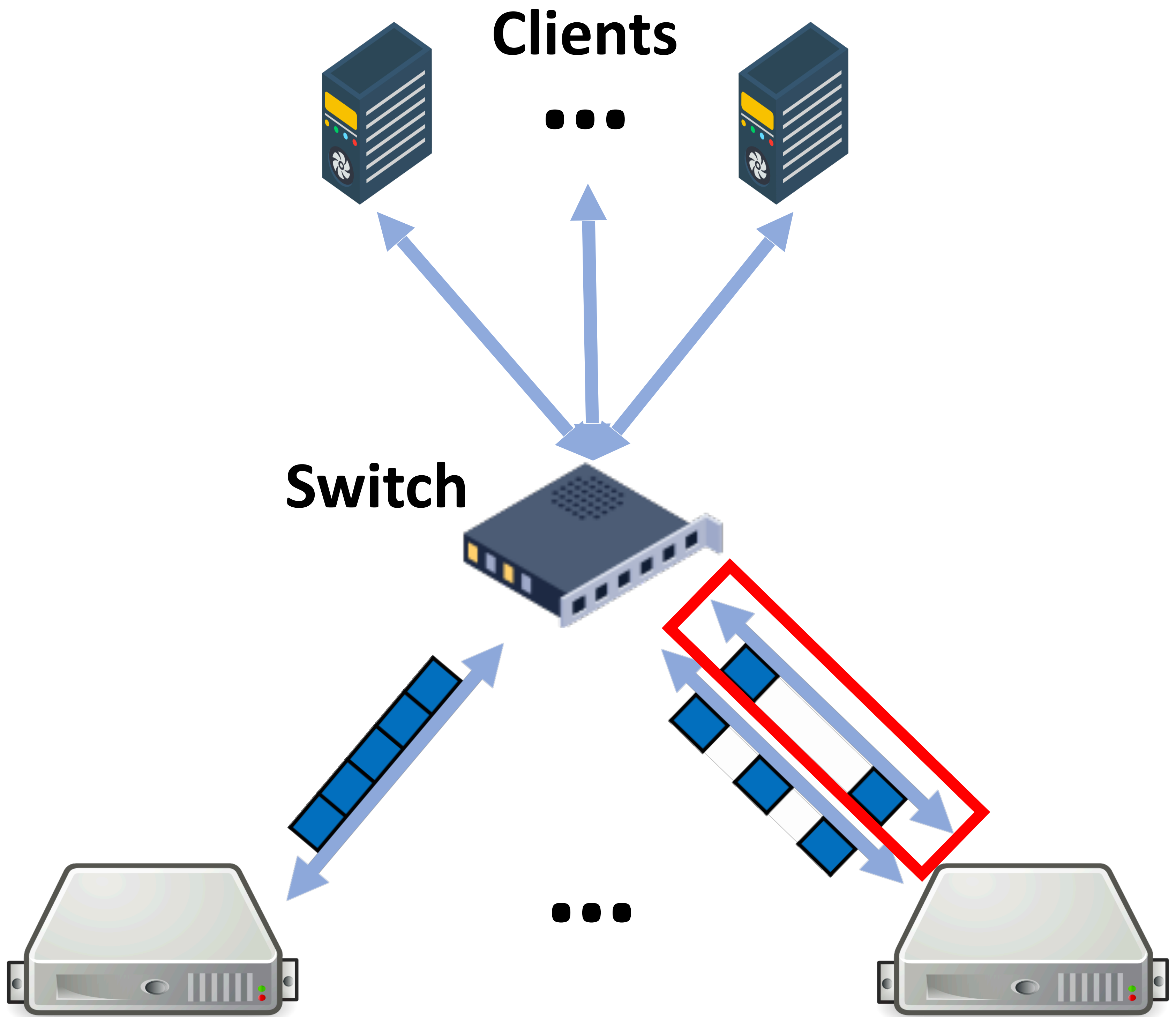
Systems like Pegasus assume the load balancer acts on a *packet level*

Common for many advanced load balancing and in-network computing apps
e.g. SwitchKV [NSDI ’16], NetCache [SOSP’17]

...but...

Most datacenter traffic is TCP-based!

Load balancer can’t just redirect traffic on packet level



TCP migration to the rescue?

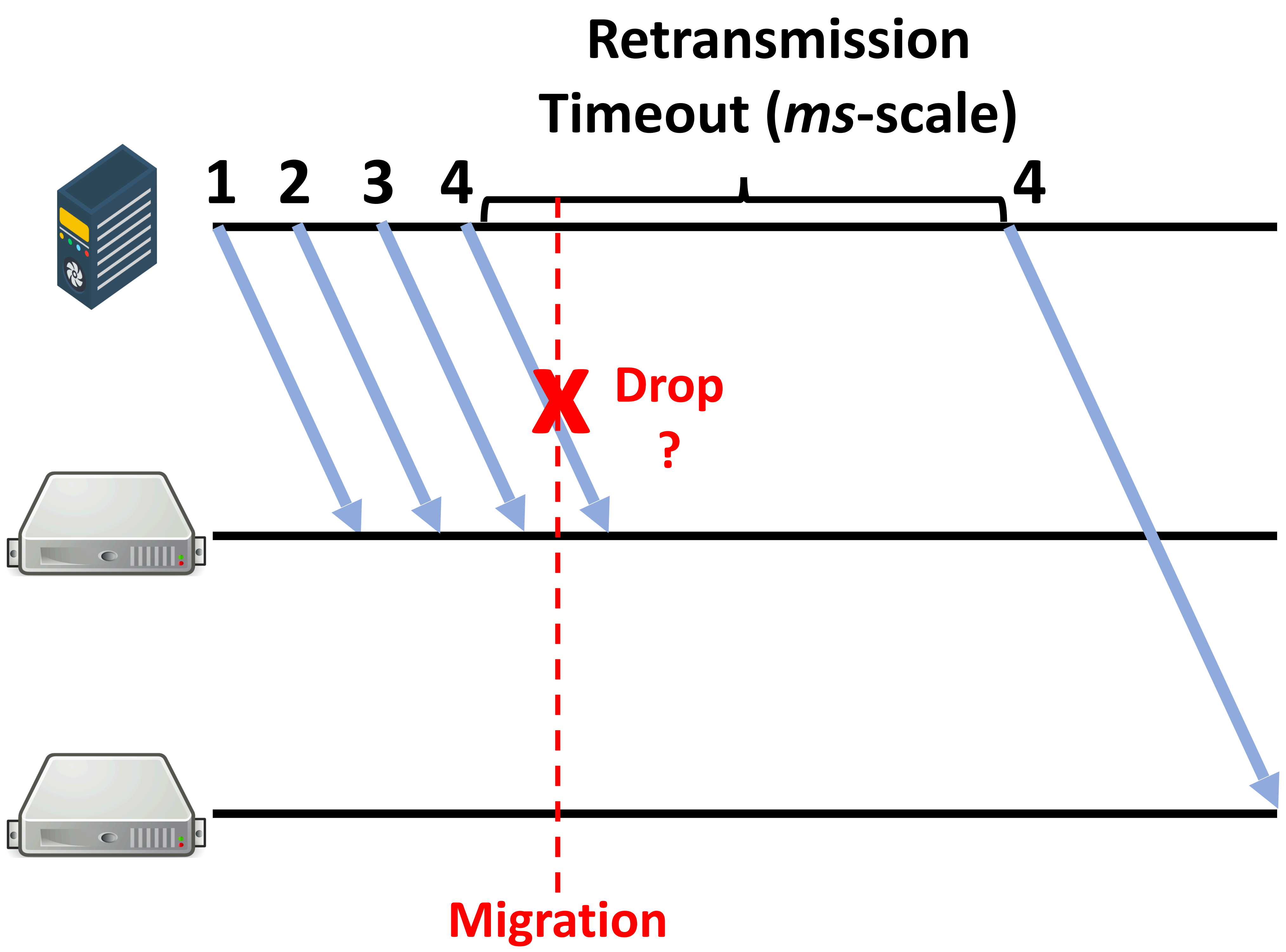
What if we could migrate an active TCP connection between servers?

We could apply the benefits of approaches like Pegasus
to more real applications

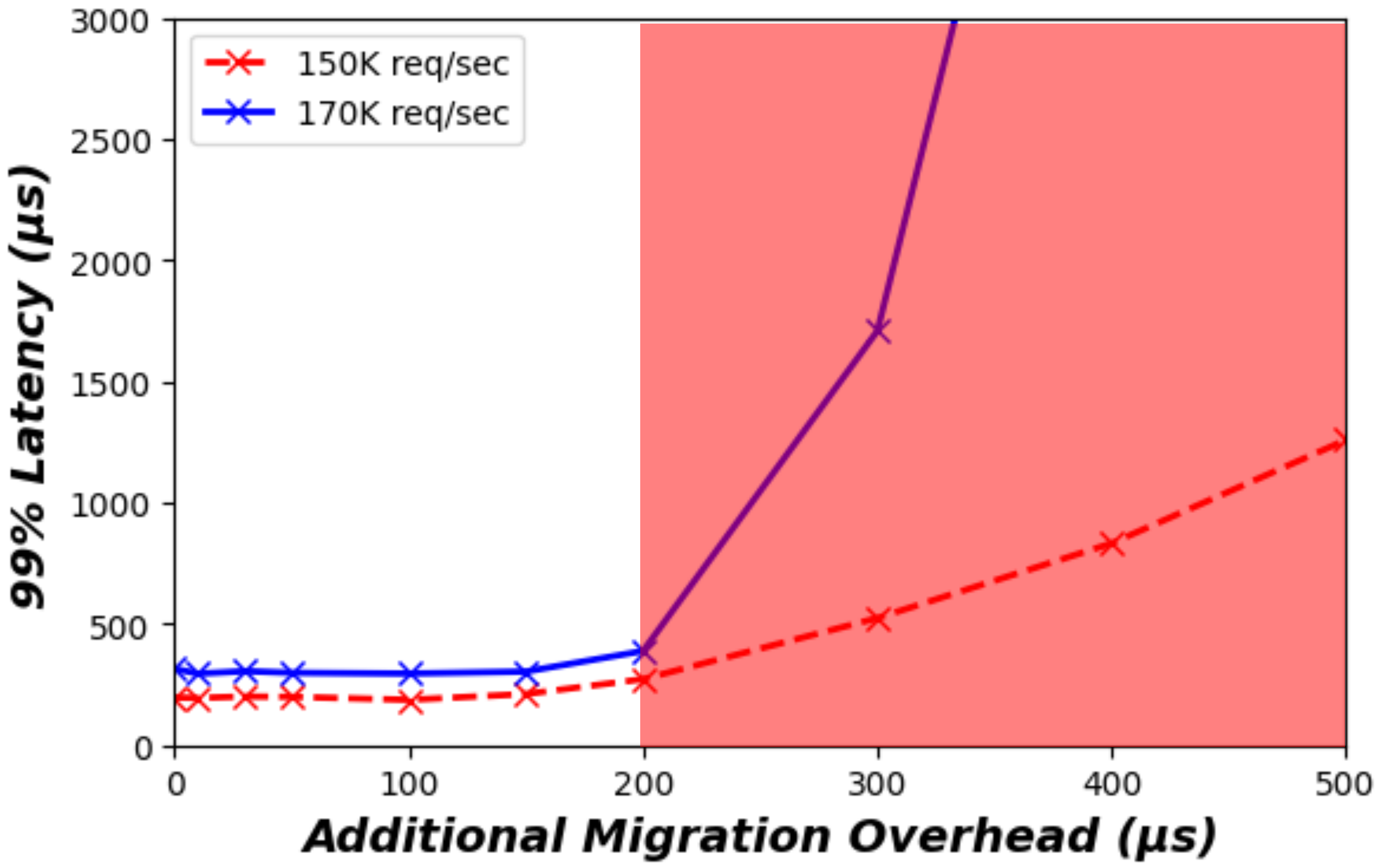
TCP migration is an old idea!

- M-TCP [1997]
- TCP Migrate [2000]

Disruptive or slow migration can make things worse!



Disruptive Migration

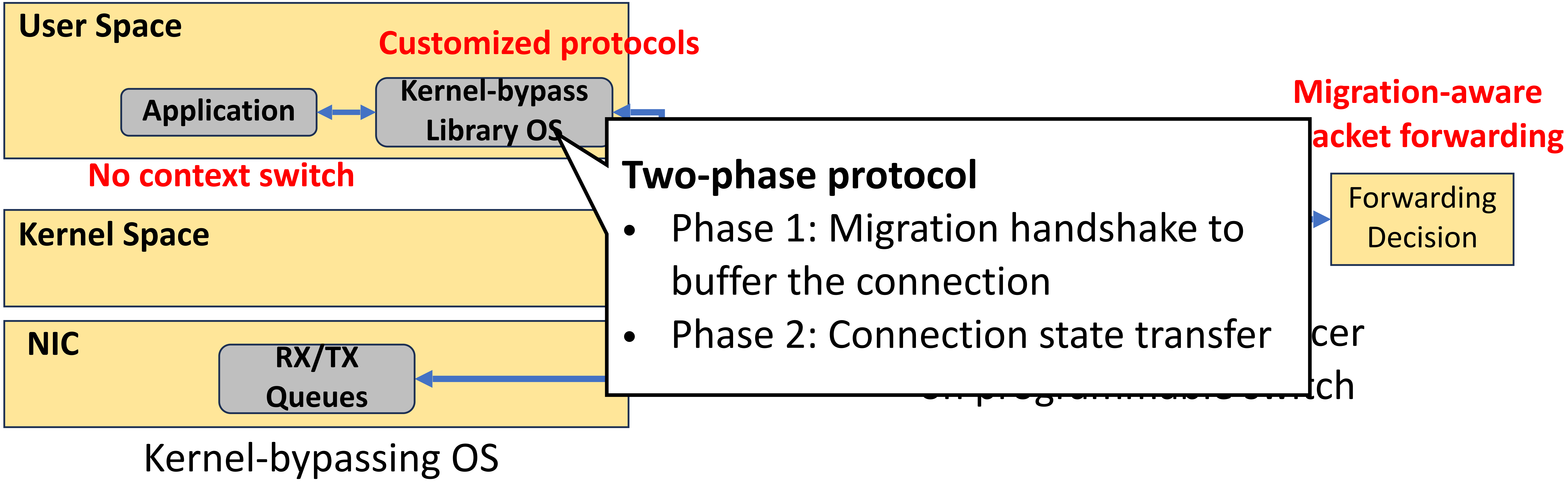


Slow Migration

Capybara: µs-scale client-transparent TCP migration

[I. Choi et al, Capybara: Microsecond-Scale Live TCP Migration, APSys'23]

Capybara: µs-scale client-transparent TCP migration

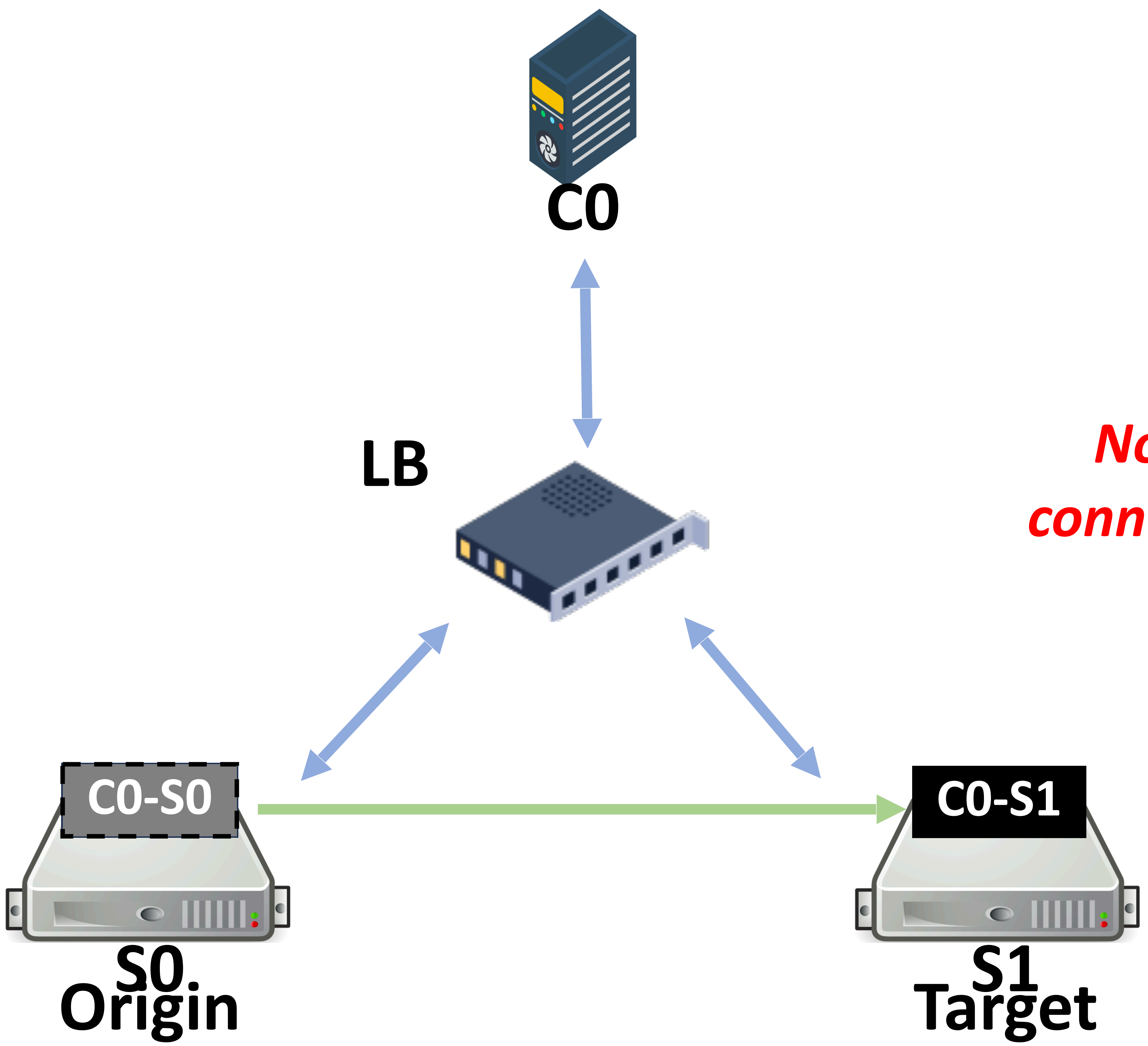


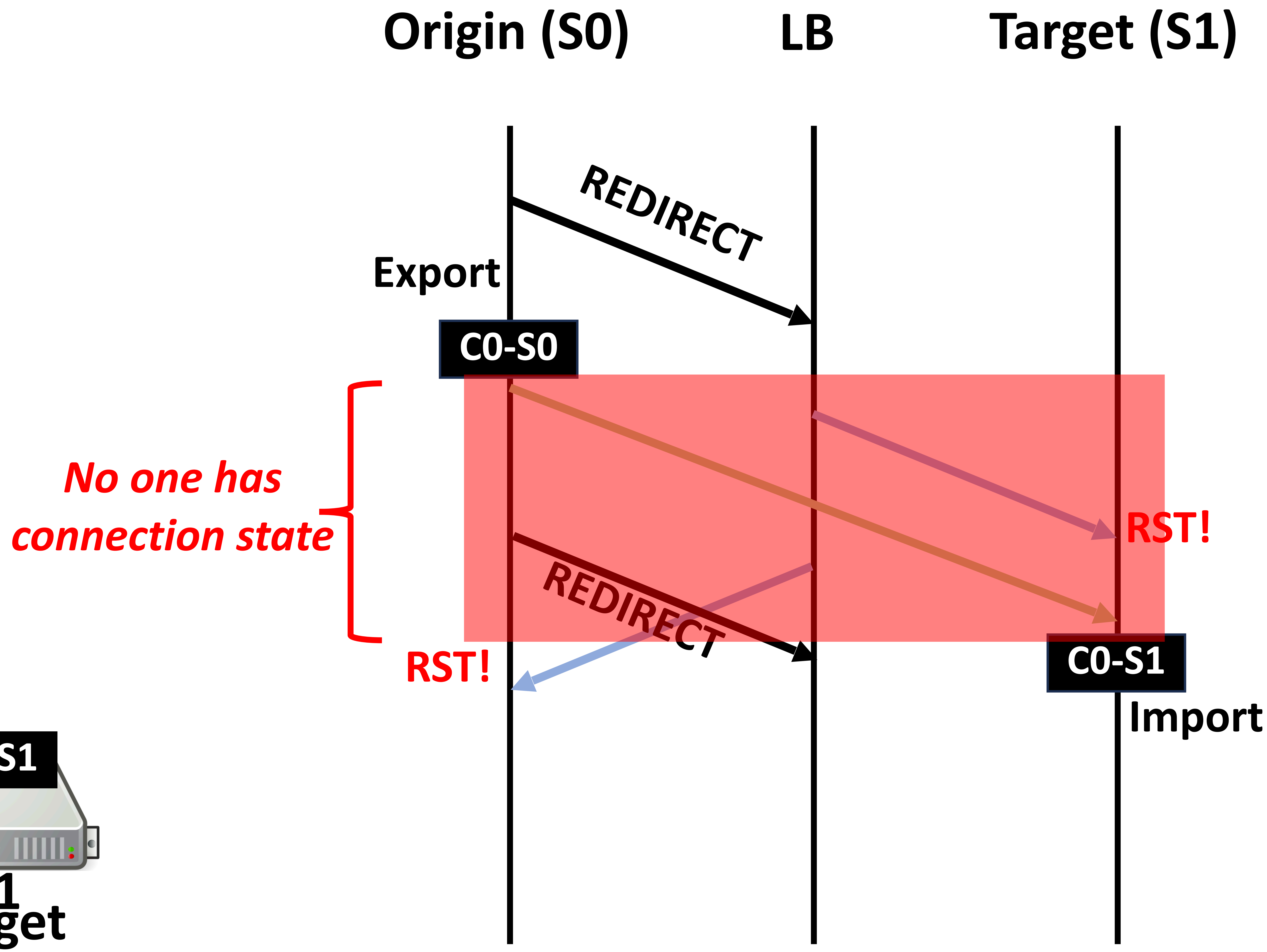
<u>Transparently</u> migrate a live TCP connection within single-digit μs

"without disconnection or blocking"

[I. Choi et al, Capybara: Microsecond-Scale Live TCP Migration, APSys'23]

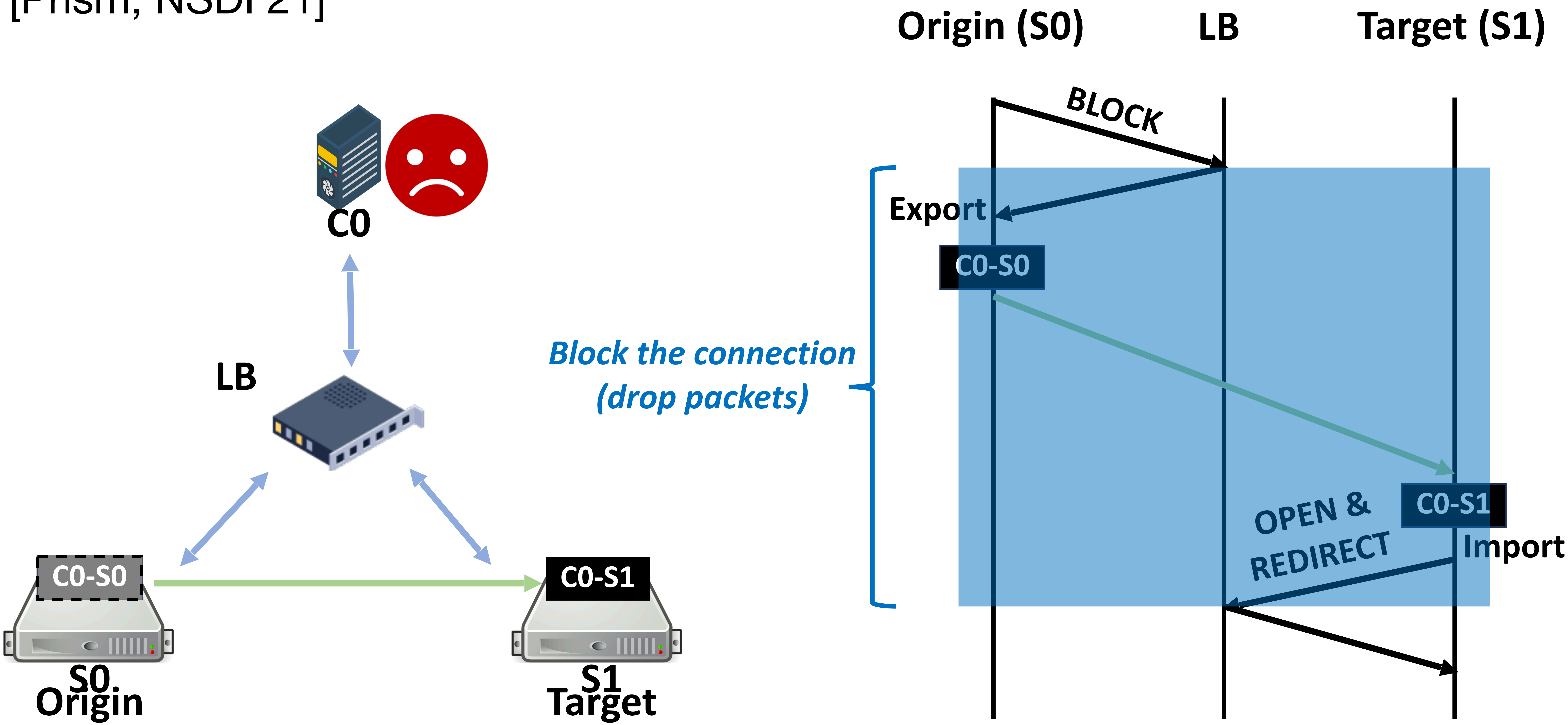
Naïve approach can reset the connection



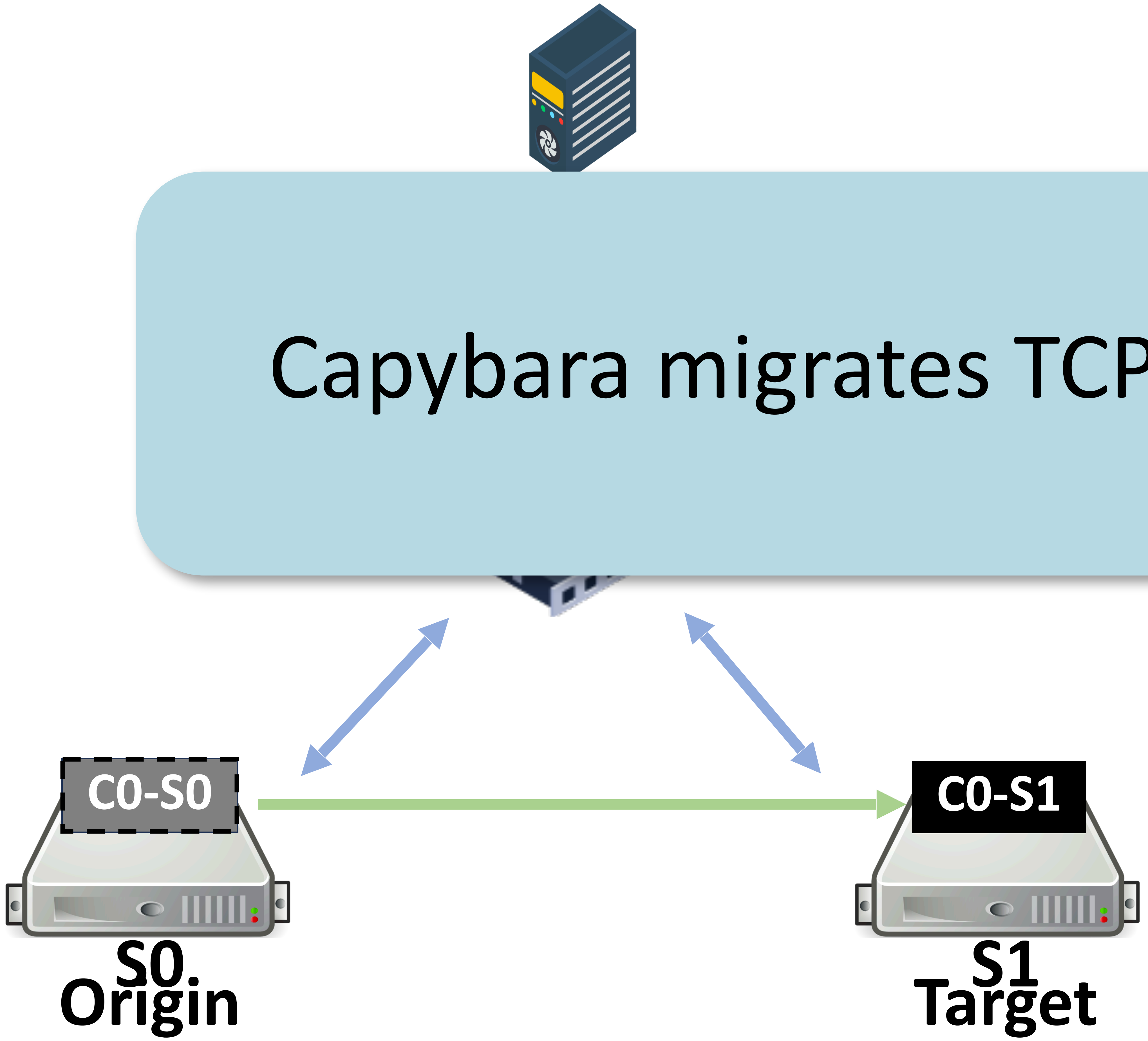


Block the connection during migration?

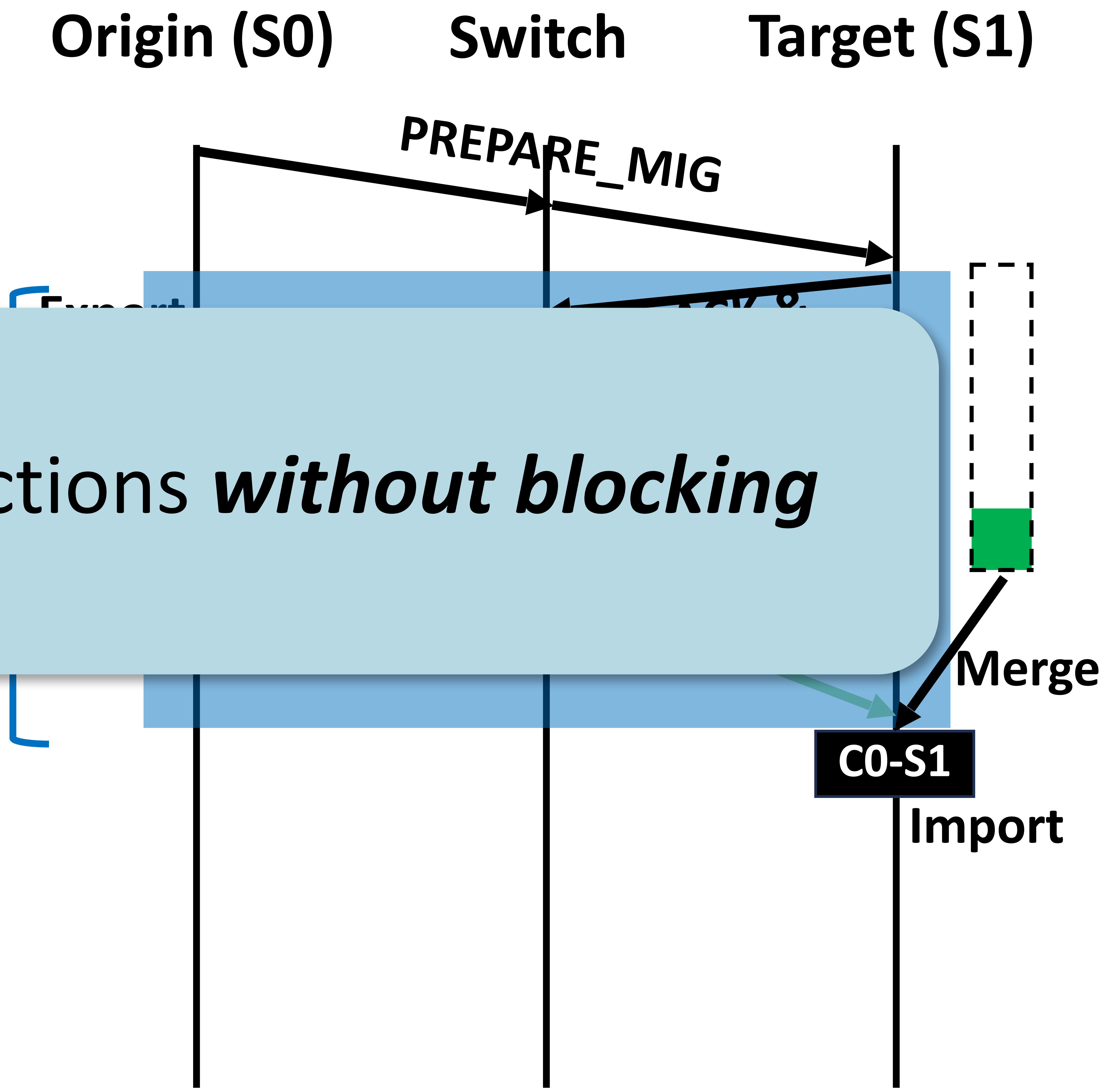
[Prism, NSDI'21]



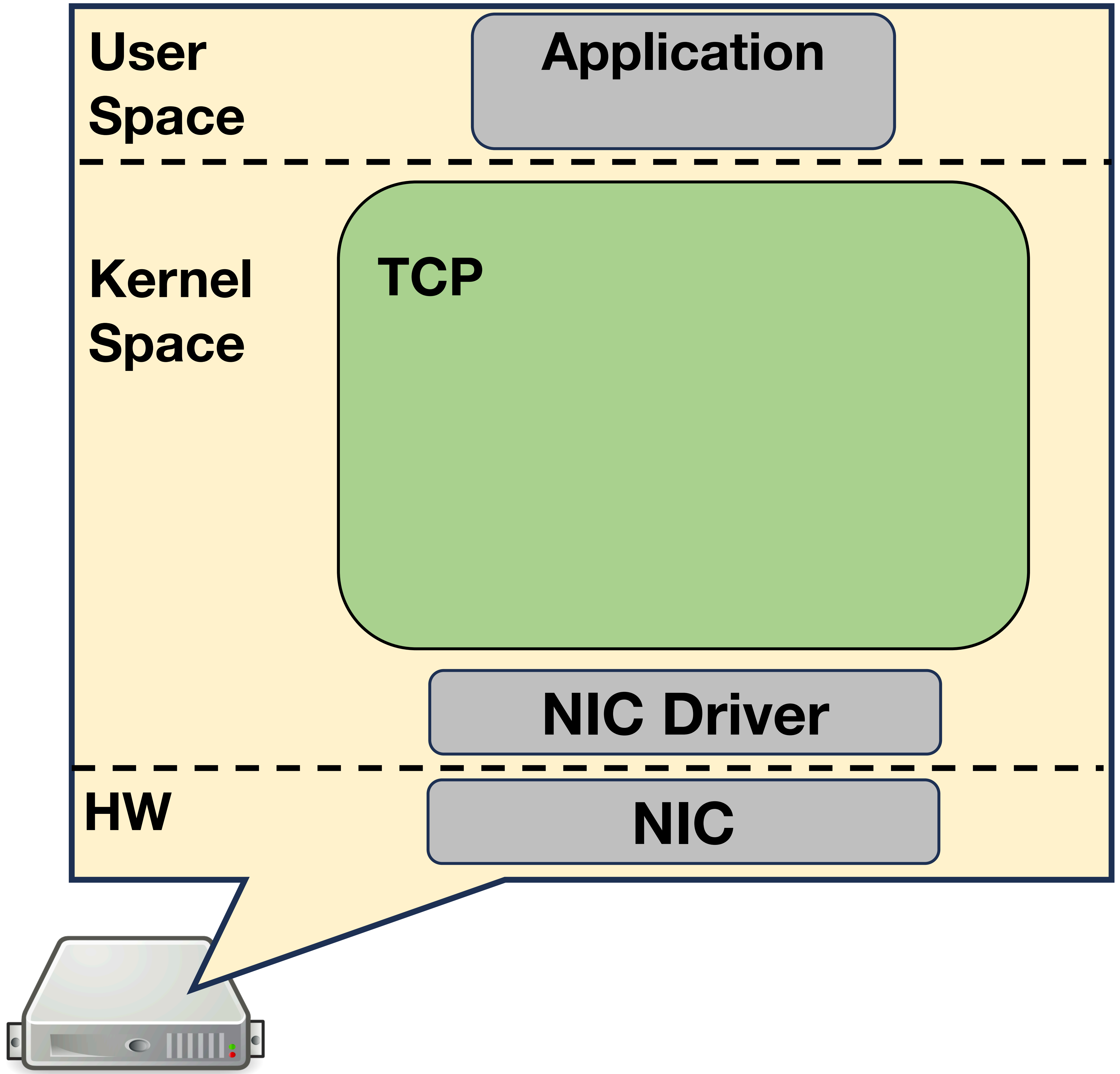
Capybara approach: transient packet buffering



Capybara migrates TCP connections ***without blocking***



Server-side architecture



Standard (Linux-based) Server

Server-side architecture

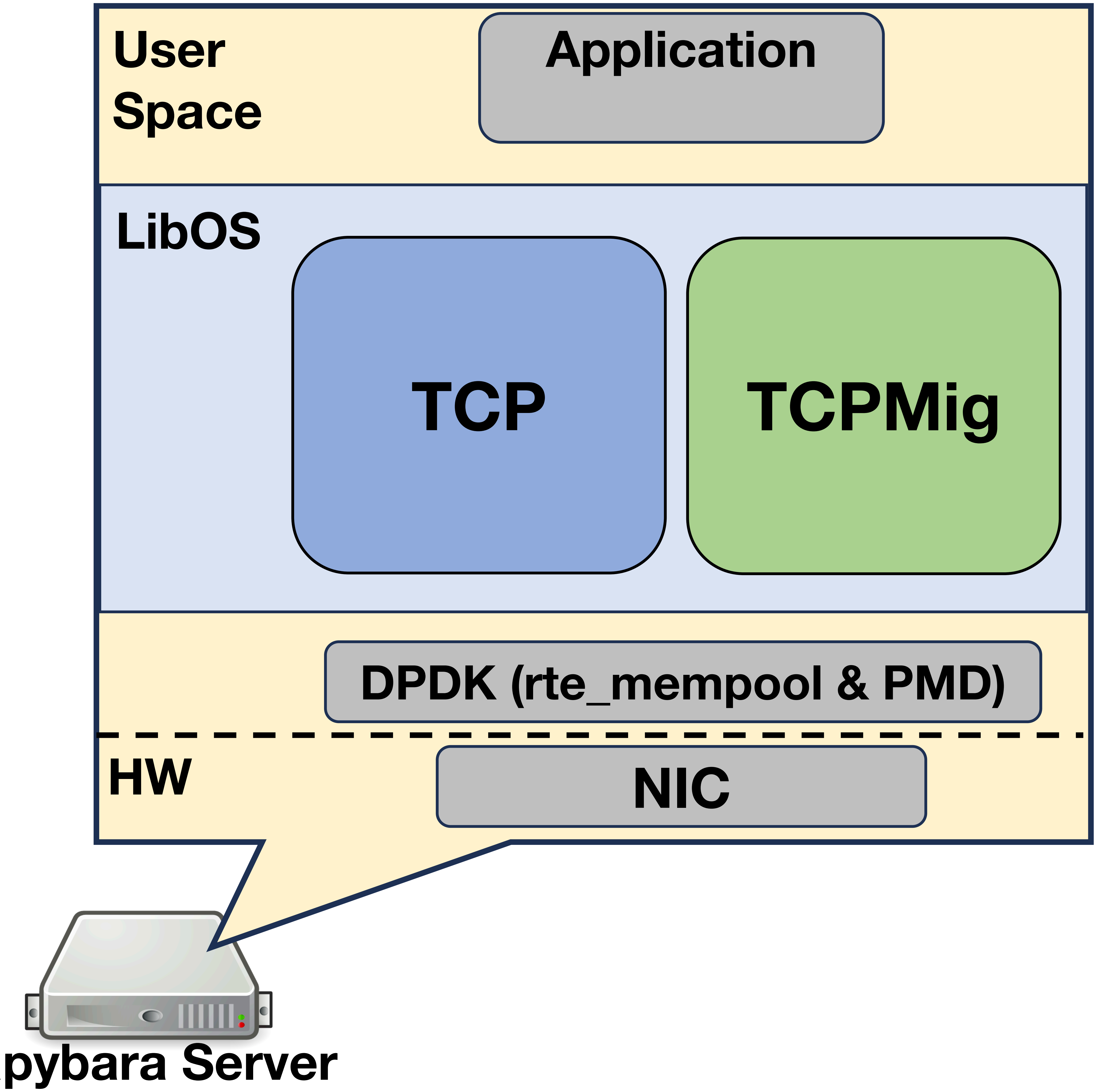
Implements TCP migration protocol in Demikernel LibOS [SOSP '21]

TCP

- Tracking TCP receive queue length
- TCP state serialization/deserialization

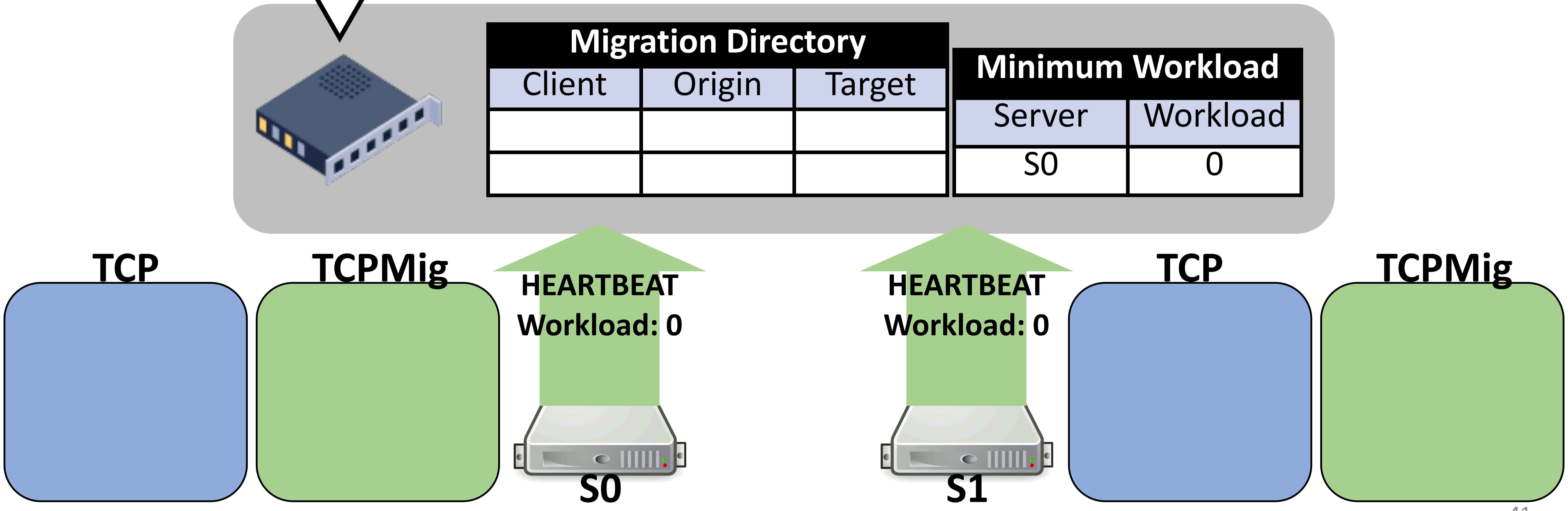
TCPMig

- Manage ongoing migration instances
- Transient packet buffering

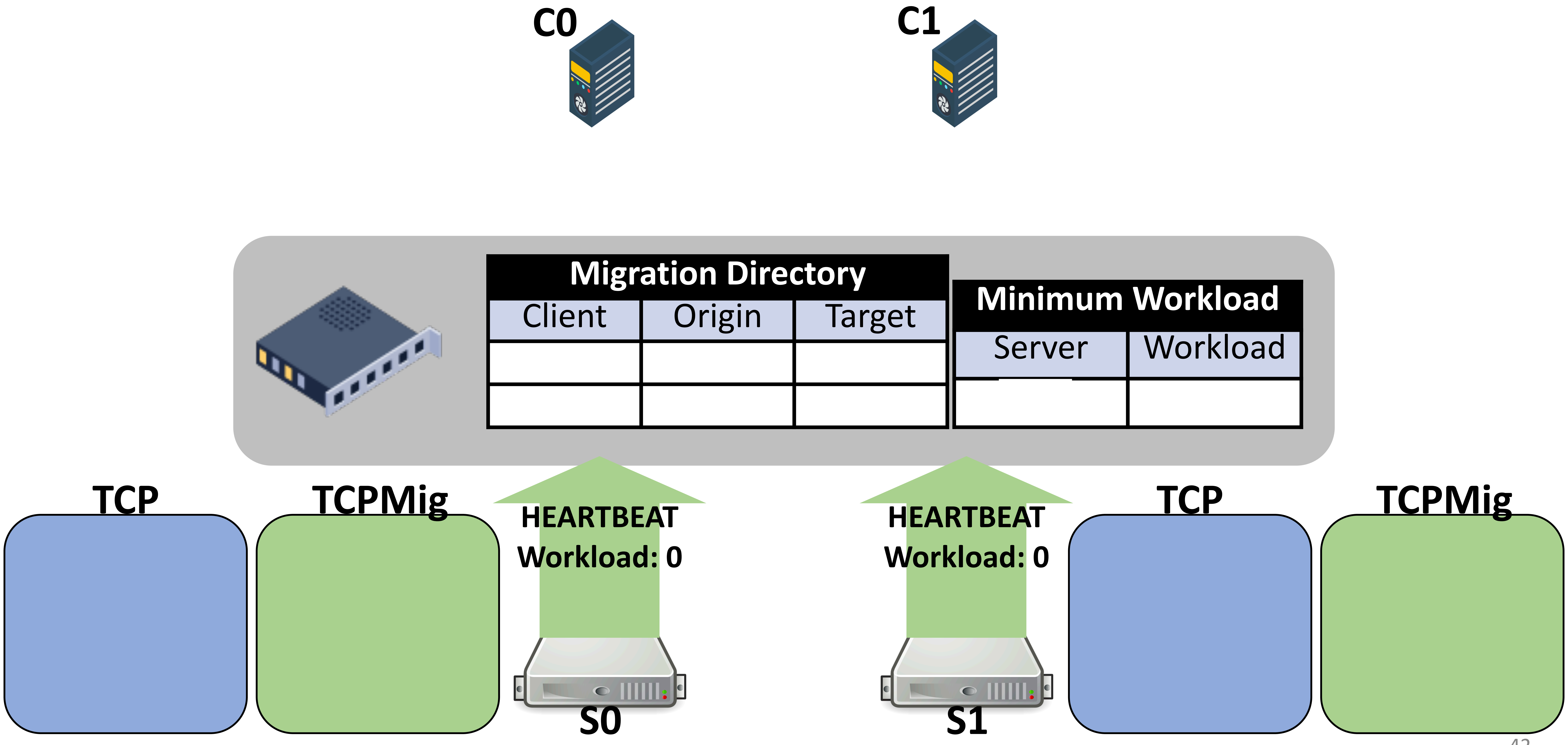


Switch architecture

1. Migration-aware packet forwarding



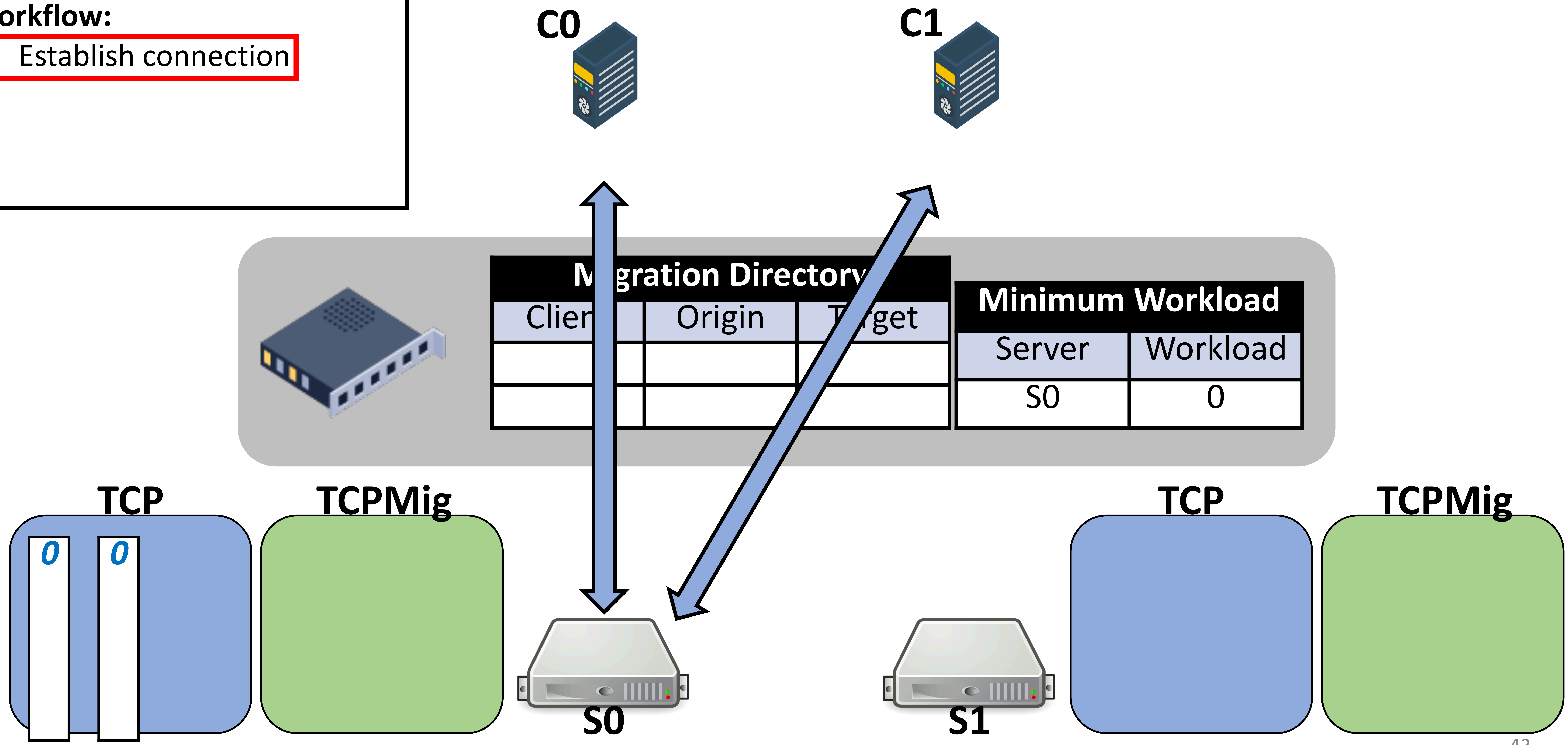
Tracking server load



Connection establishment

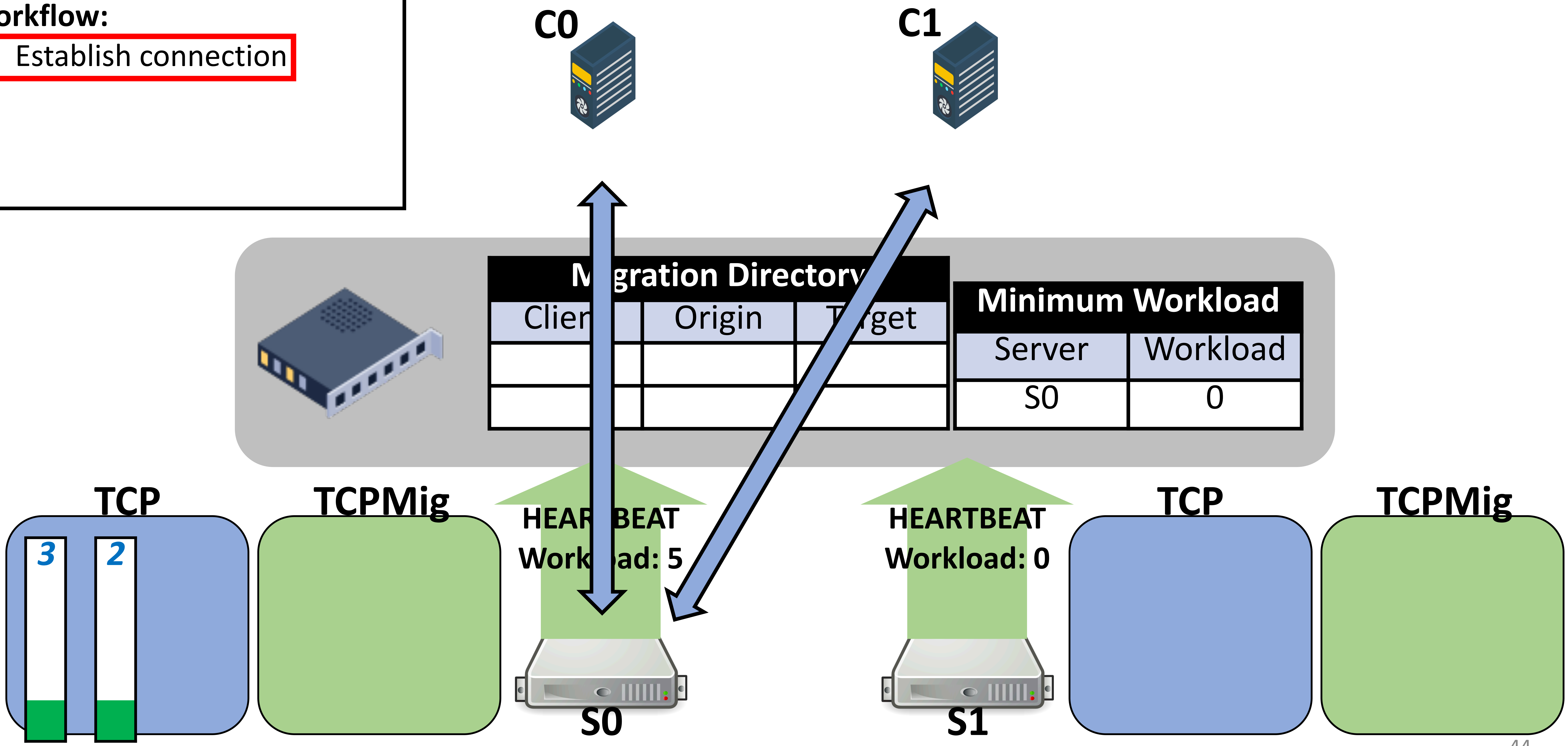
Workflow:

1. Establish connection



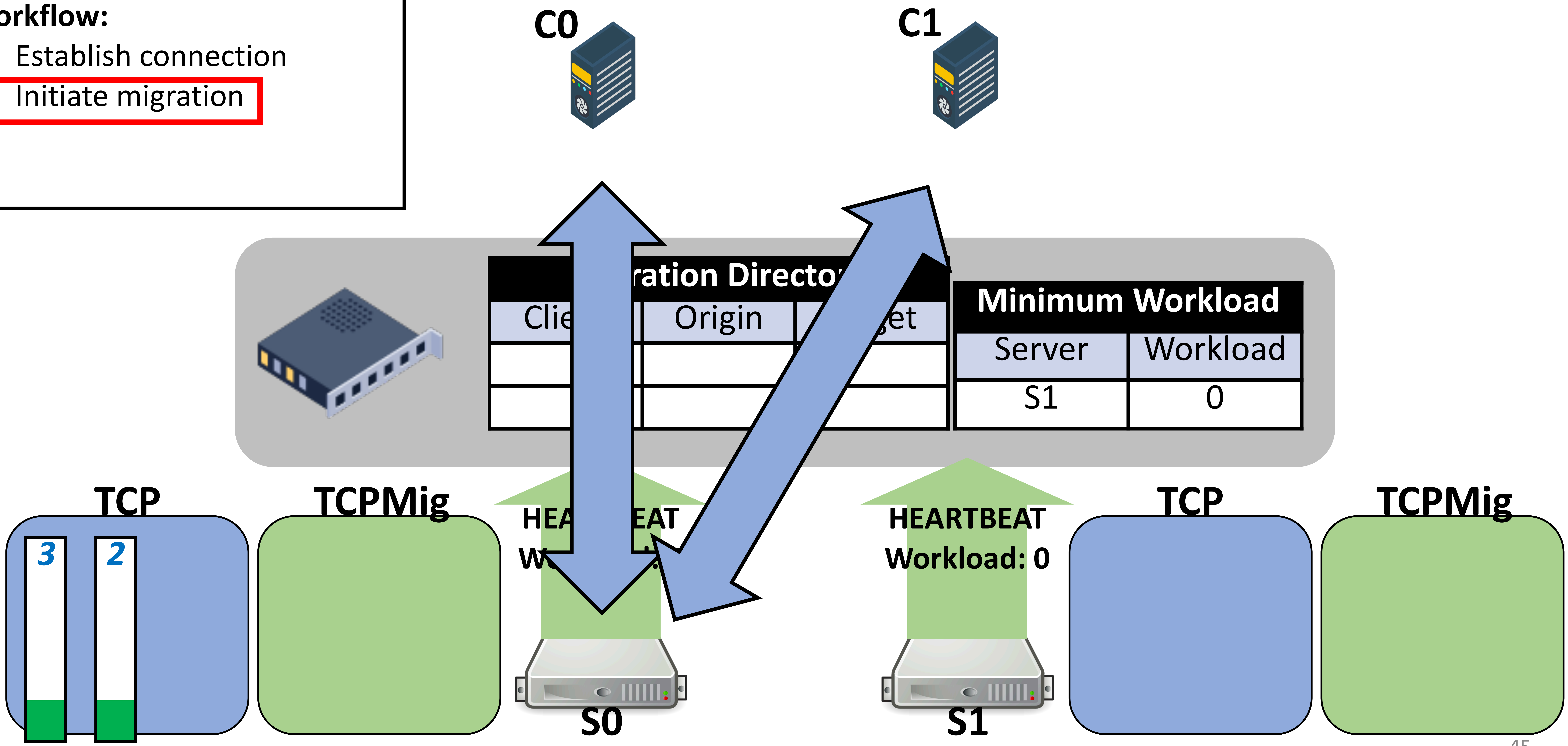
Connection establishment

Workflow:
1. Establish connection



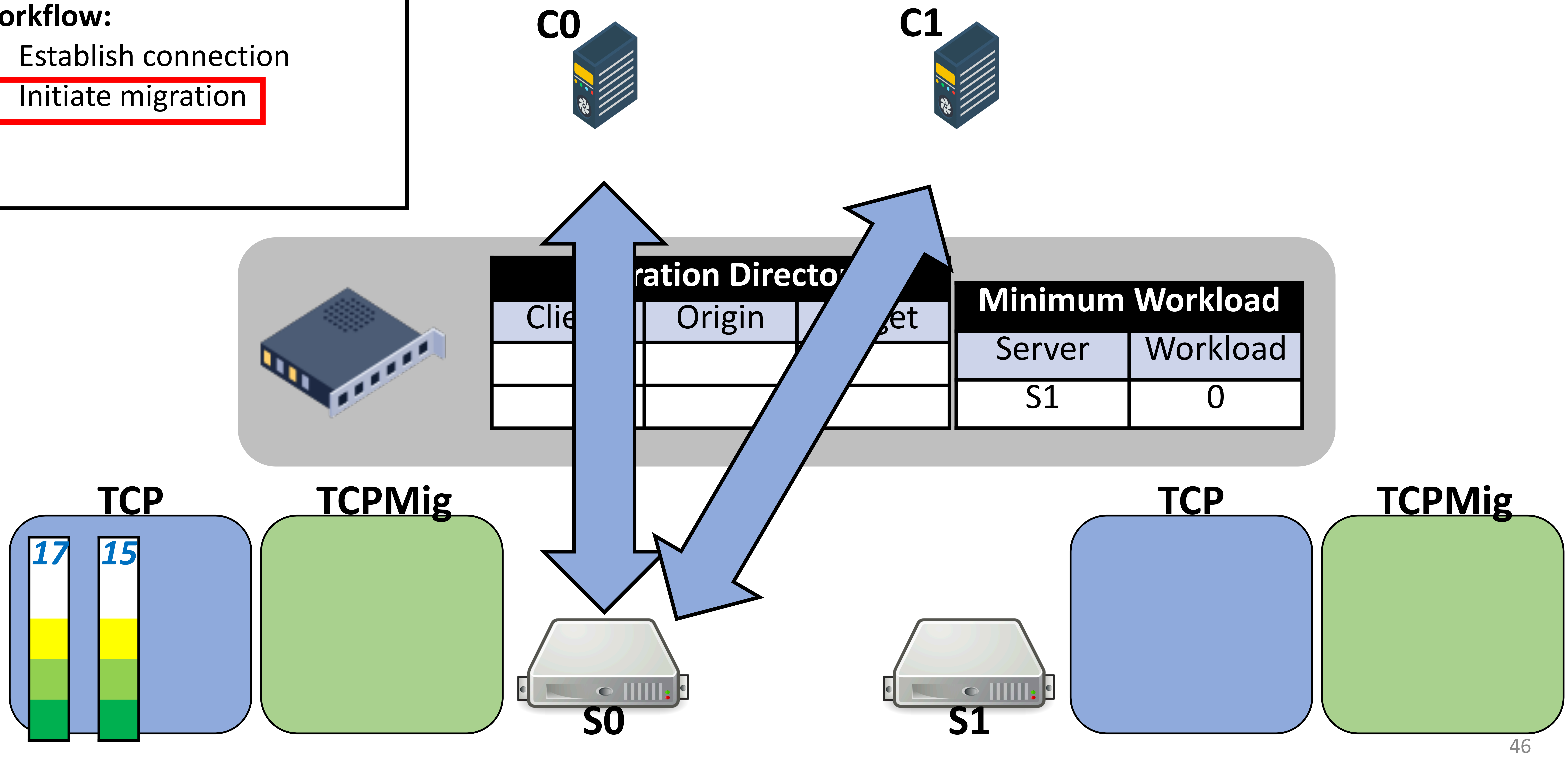
Server overload detected

Workflow:
1. Establish connection
2. Initiate migration



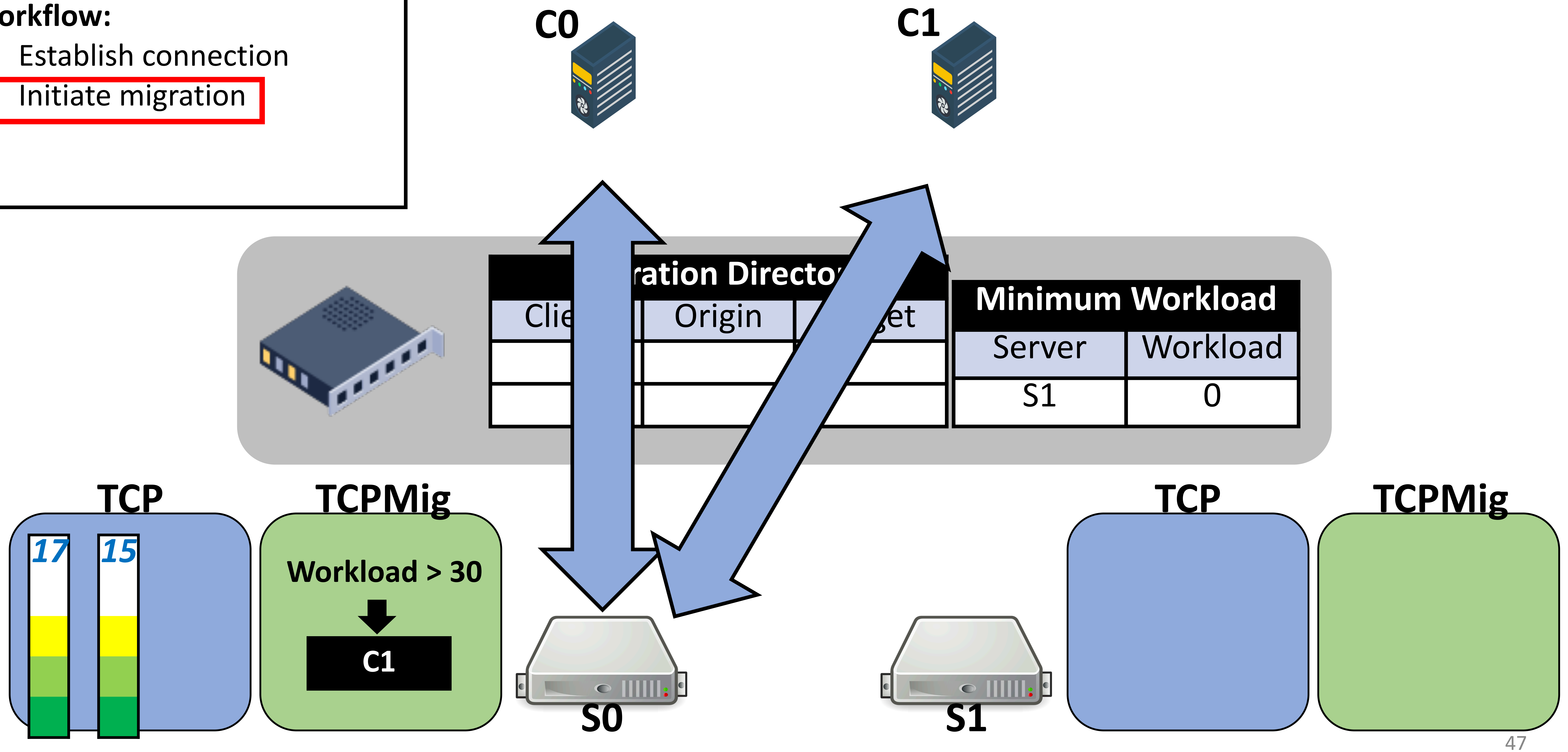
Server overload detected

Workflow:
1. Establish connection
2. Initiate migration



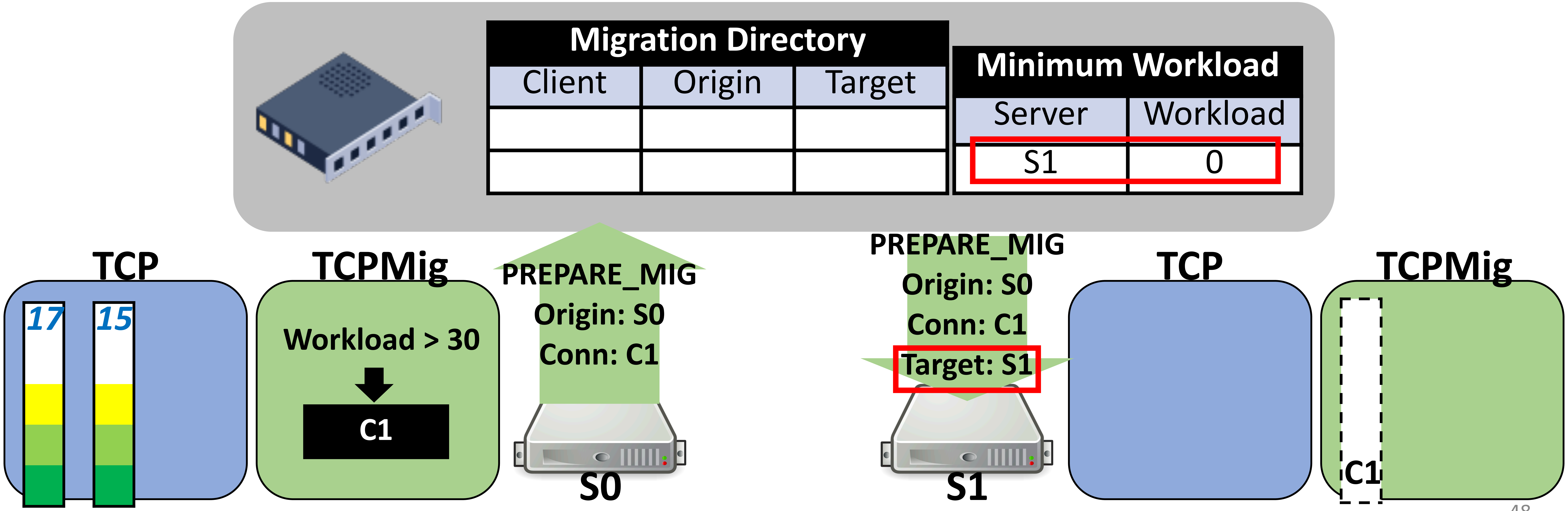
Server overload detected

Workflow:
1. Establish connection
2. Initiate migration



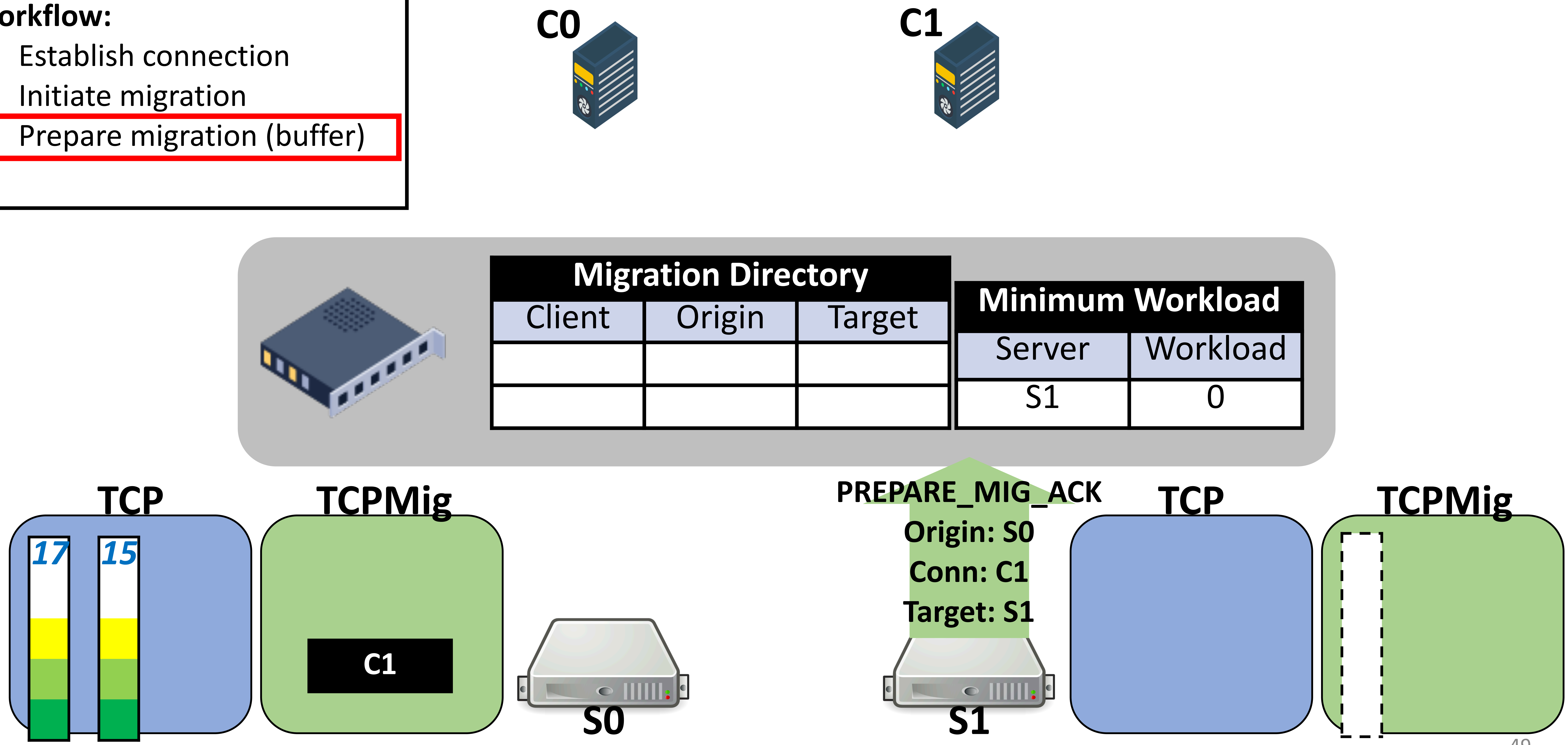
Phase 1: Prepare migration

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)



Phase 1: Prepare migration

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)

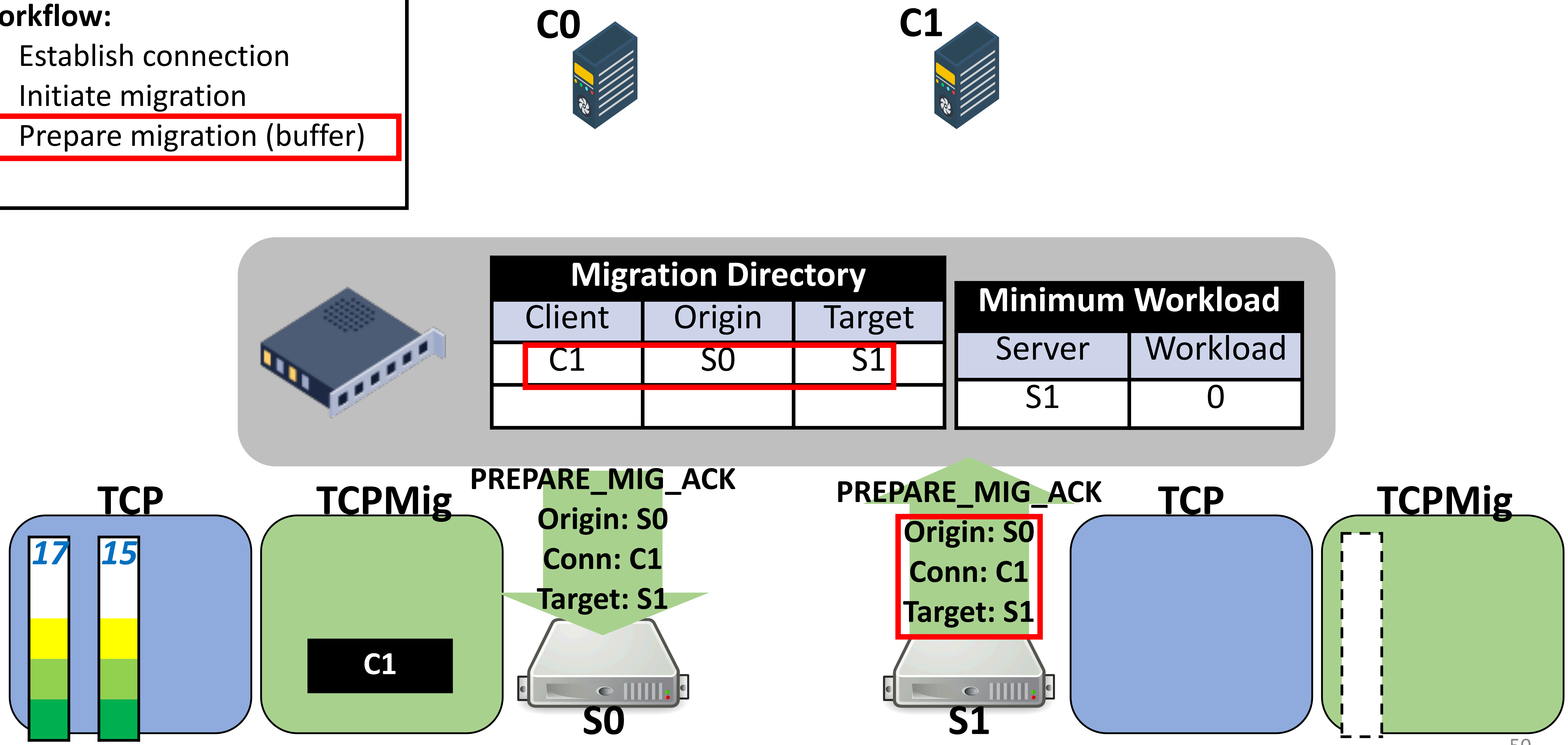


Migration Directory		
Client	Origin	Target

Minimum Workload	
Server	Workload
S1	0

Phase 1: Prepare migration

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)

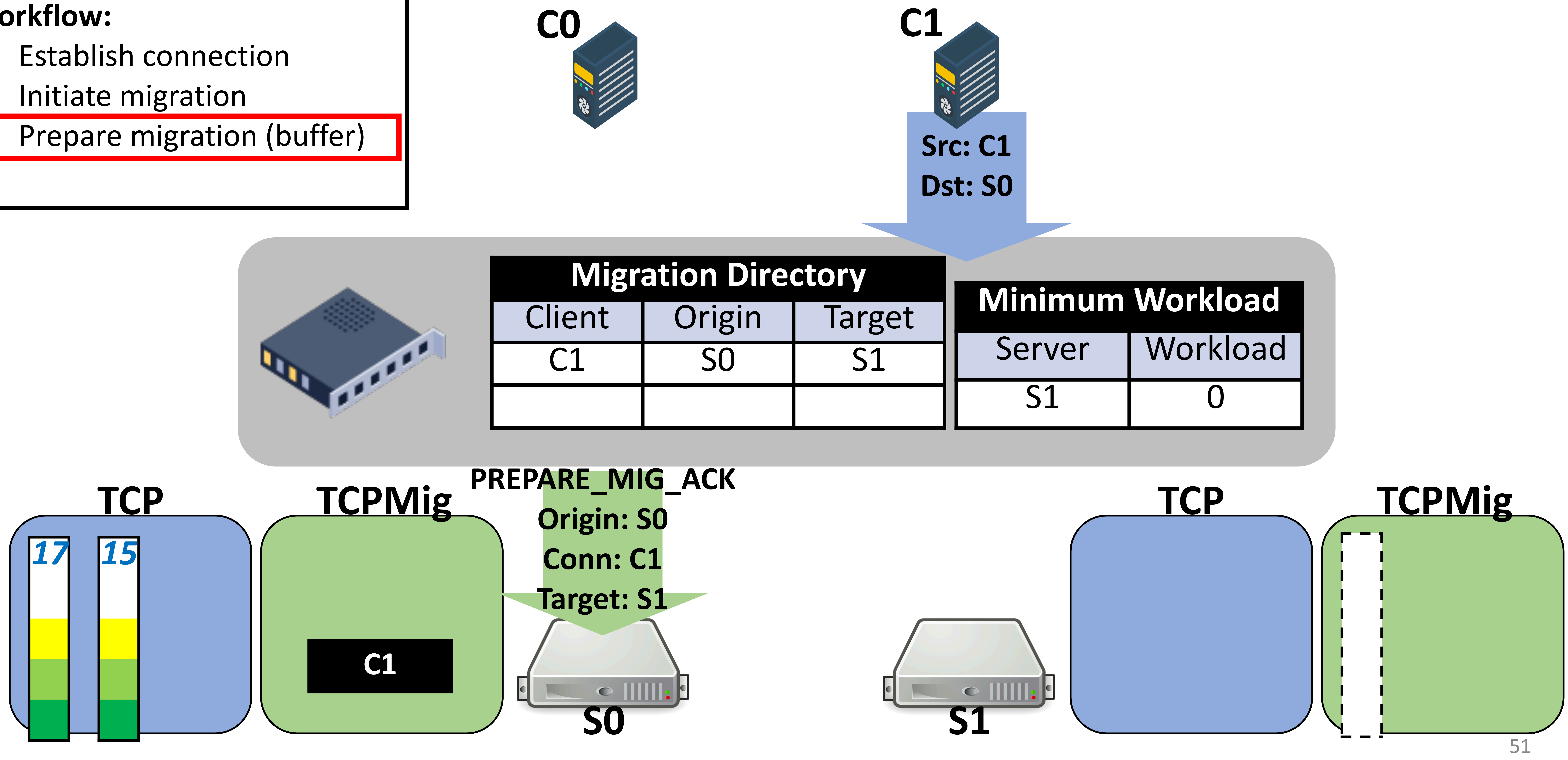


Migration Directory		
Client	Origin	Target
C1	S0	S1

Minimum Workload	
Server	Workload
S1	0

Message buffering

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)

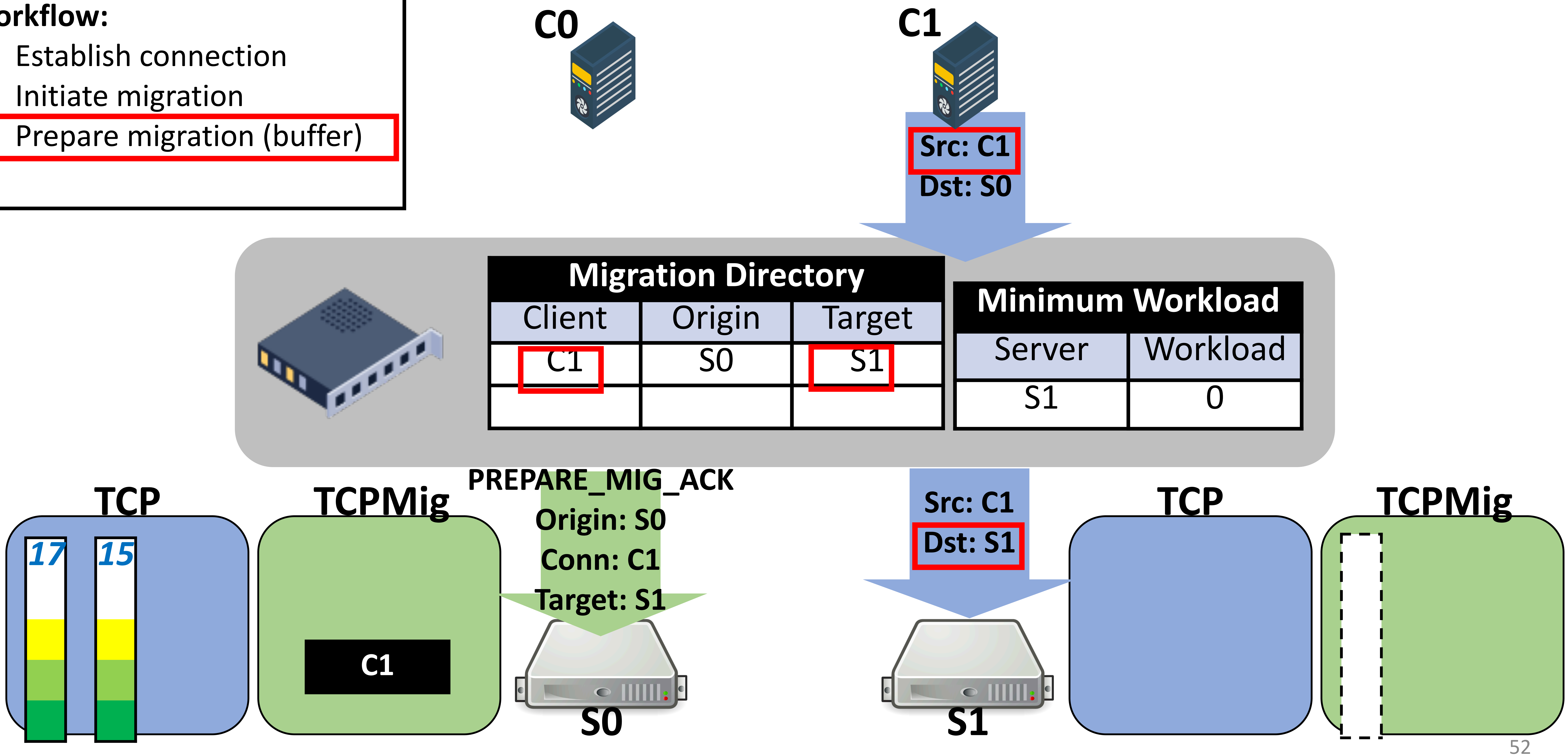


Migration Directory		
Client	Origin	Target
C1	S0	S1

Minimum Workload	
Server	Workload
S1	0

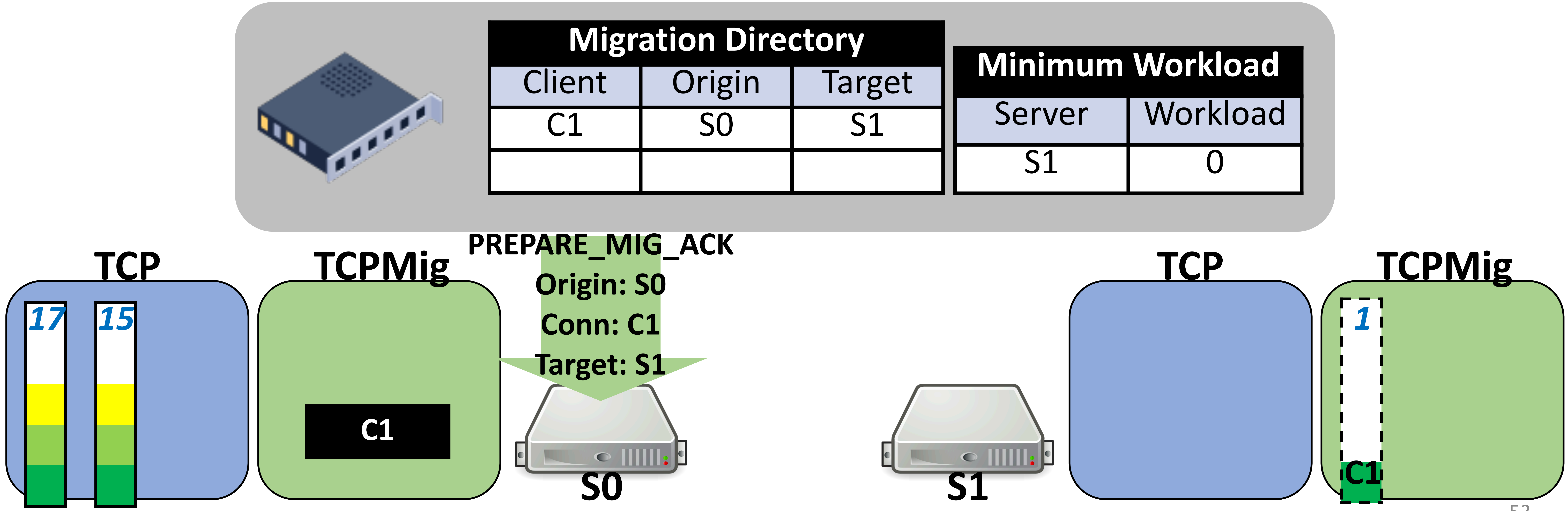
Message buffering

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)



Message buffering

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)



Migration Directory		
Client	Origin	Target
C1	S0	S1

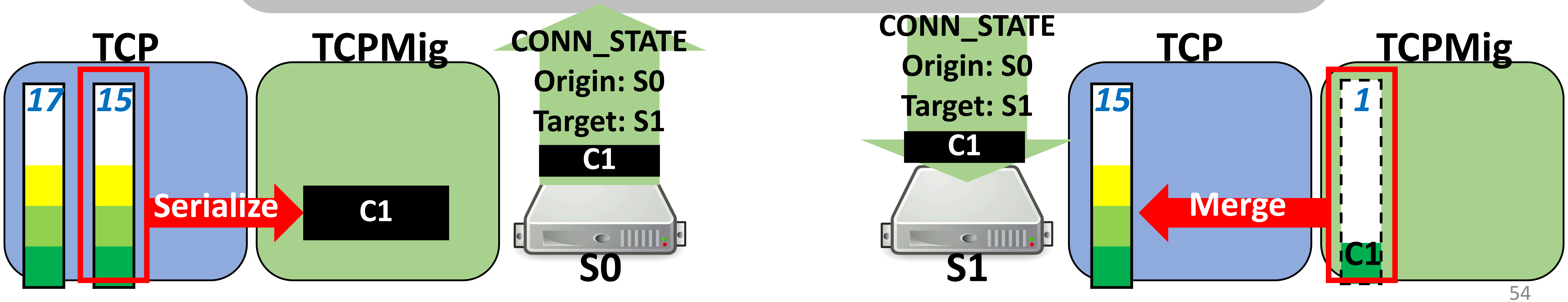
Minimum Workload	
Server	Workload
S1	0

Phase 2: state transfer

Workflow:
1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)
4. Transfer connection state

Migration Directory		
Client	Origin	Target
C1	S0	S1

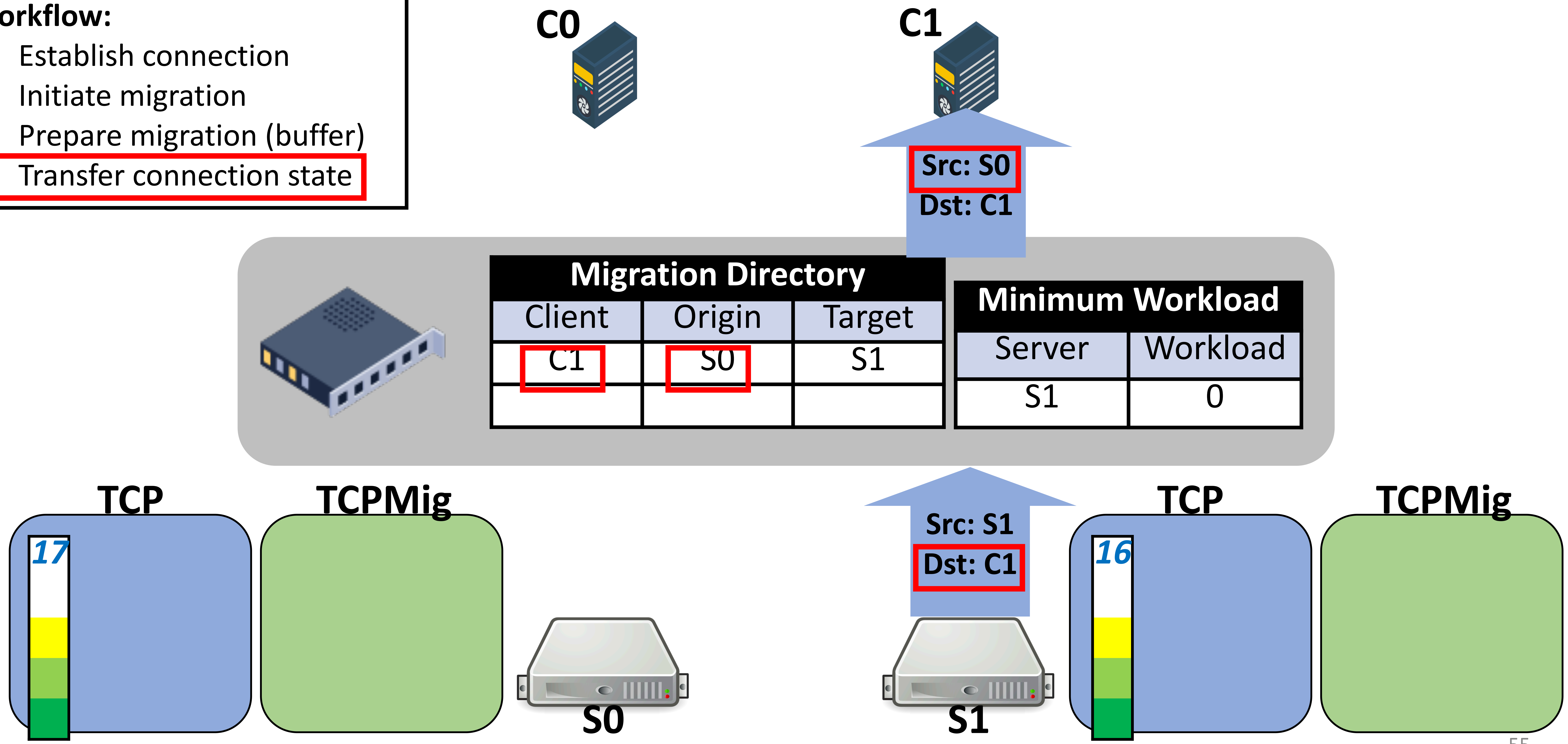
Minimum Workload	
Server	Workload
S1	0



Rewriting response traffic

Workflow:

1. Establish connection
2. Initiate migration
3. Prepare migration (buffer)
4. Transfer connection state



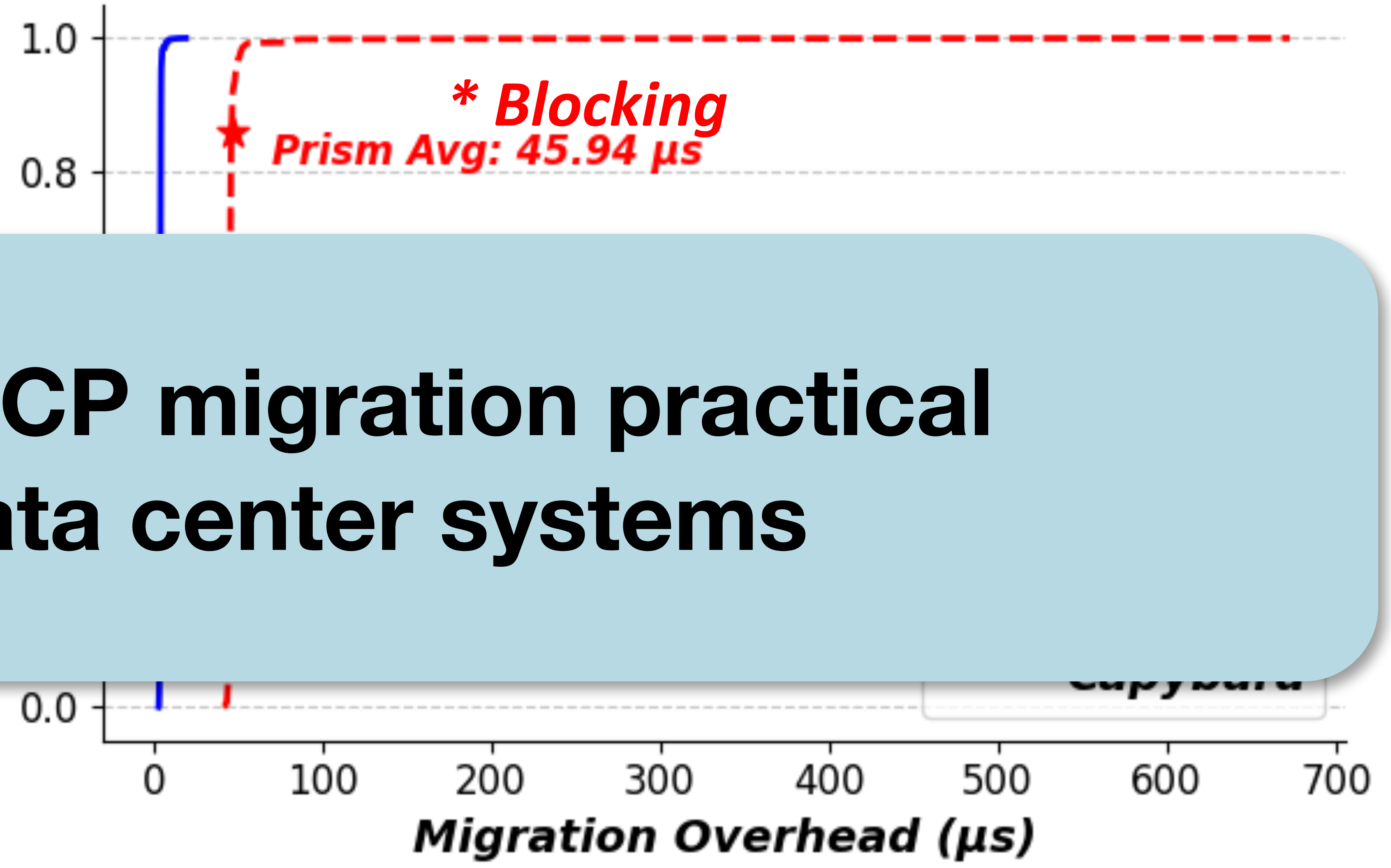
Migration Directory		
Client	Origin	Target
C1	S0	S1

Minimum Workload	
Server	Workload
S1	0

Capybara enables stable, low latency migration

Microbenchmark compares Prism (Linux based) to Capybara

- 10,000 TCP migrations between two servers



Capybara makes TCP migration practical for μs-scale data center systems

Capybara

- Avg: ~ 3.90 μs (12x faster)
- Stable

Capybara Summary

Capybara provides TCP migration with single-digit microsecond latency using a custom kernel-bypass networking stack and accelerated load balancer

Fast TCP migration makes load balancing practical for more applications

Not just for load balancing:
also useful for migrations during server maintenance!

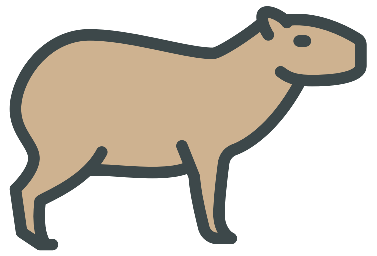
Leverages the ability of an accelerated load balancer to run custom forwarding logic at microsecond-scale

Agenda for this talk

Overview of accelerated load balancing

Three systems that enable new functionality with accelerated load balancing

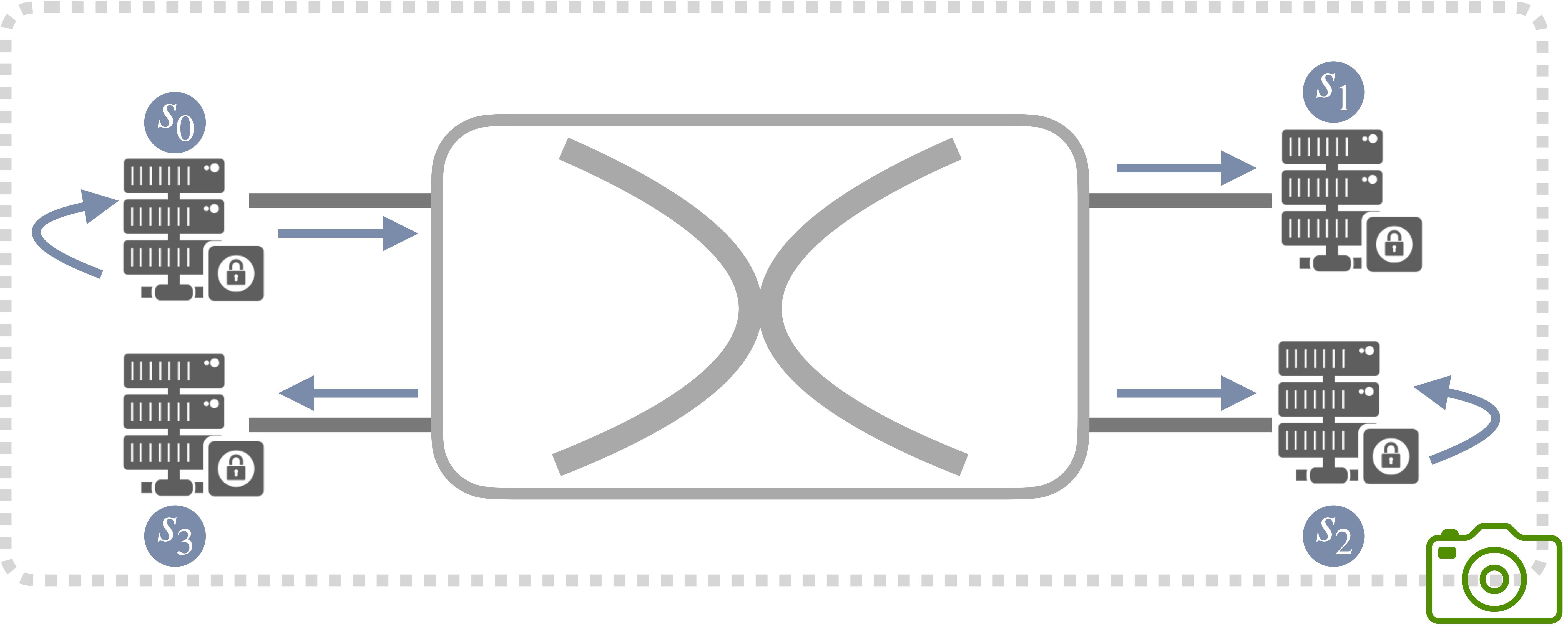
Pegasus: balancing skewed workloads in distributed storage



Capybara: live migration of active TCP connections at μs-scale

Beaver: using load balancers to take practical persistent checkpoints

Distributed checkpointing is a classic problem

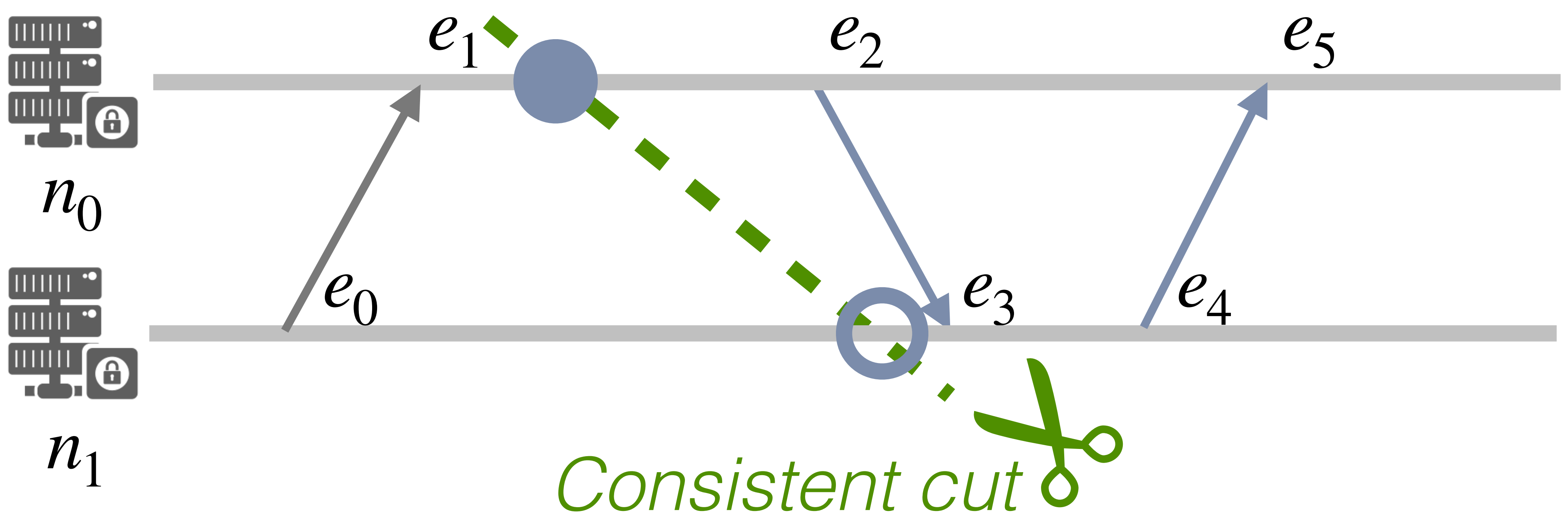


- **Events** (message send/receive, computation step…) occur distributedly
- **States** associated with the task spread across machines

- A **consistent, global view of states** is helpful
  - Checkpointing and failure recovery
  - Network telemetry
  - Deadlock detection
  - Debugging of distributed software
  - …

…with a classic solution

e.g. Chandy-Lamport [TOCS'85]

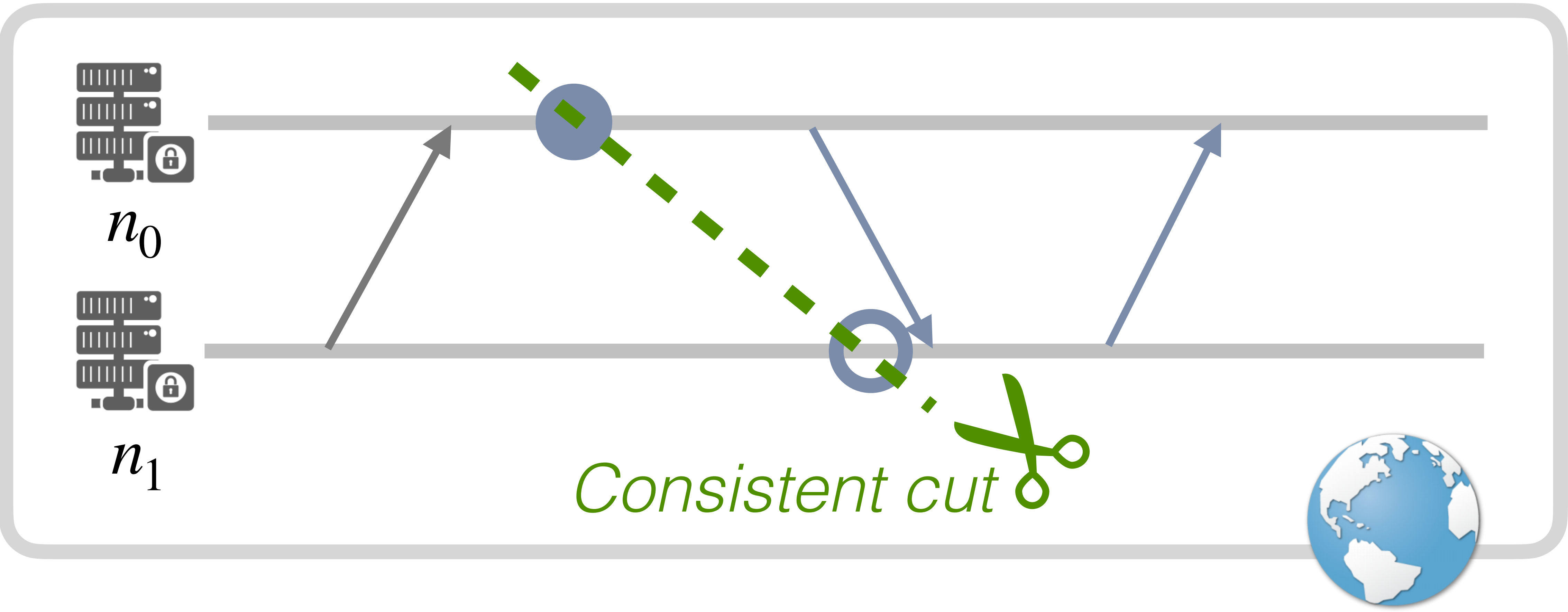


1. Initiate snapshot out-of-band
2. Mark outbound messages post-snapshot
3. Trigger snapshot (a 'cut') right before receiving a marked message
4. Collect recorded states after all nodes entered the snapshot

Guarantee of causal consistency

*For **any** event e in the cut, if $e' \rightarrow e$ ('happened before'), e' is in the cut.*

Classic algorithms operate in an isolated universe



'Universe' of nodes

Fundamental assumption:

The set of participants are ***closed*** under causal propagation.

*Unfortunately, today's cloud services are **not so utopian**!*

This assumption rarely matches reality

Modular services

Instrumentation constraints

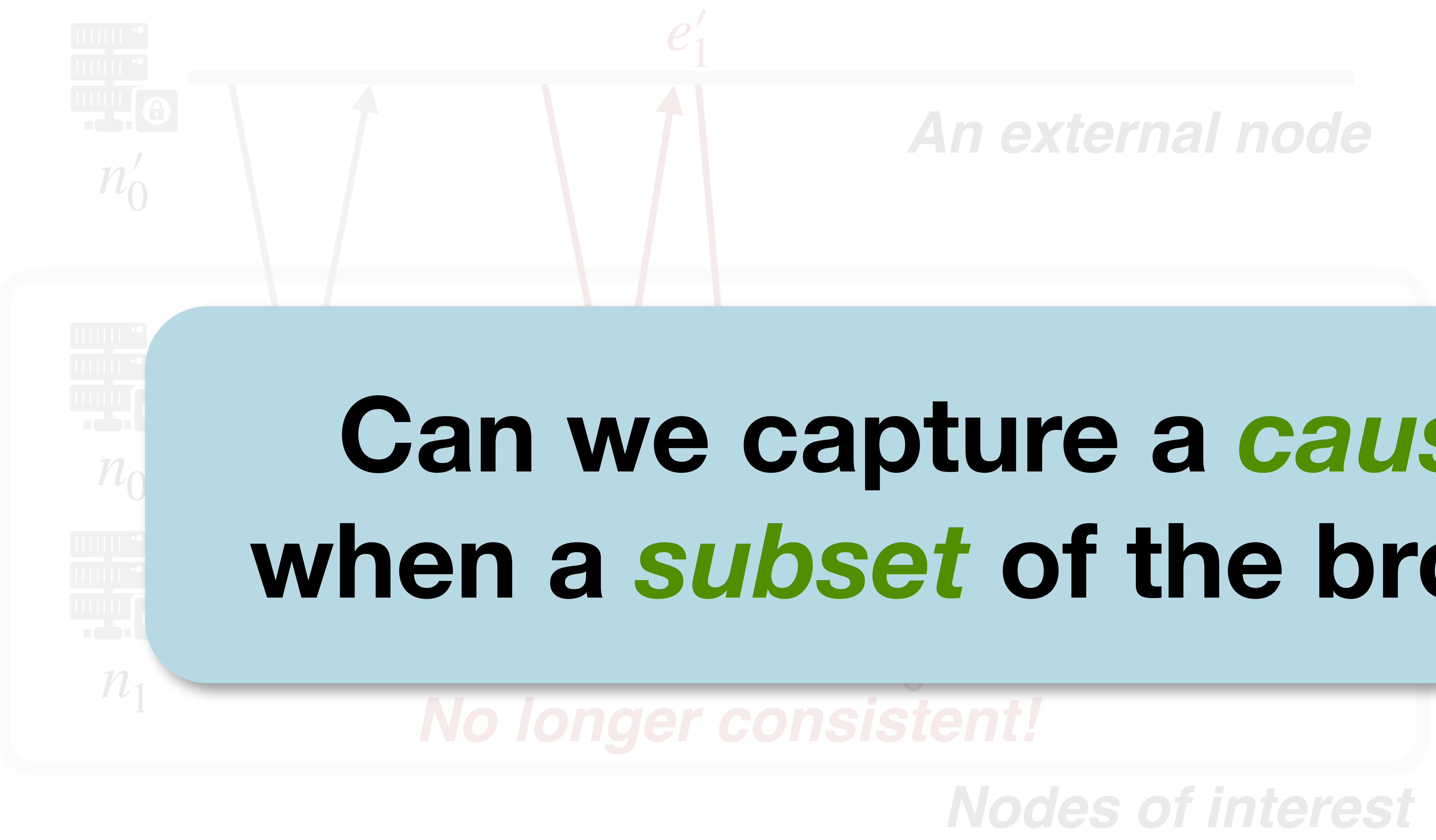
Costs and overheads

Hidden causality due to human

Not always realistic to assume ***zero interaction*** with the external world

Nor practical to instrument ***all involved processes***

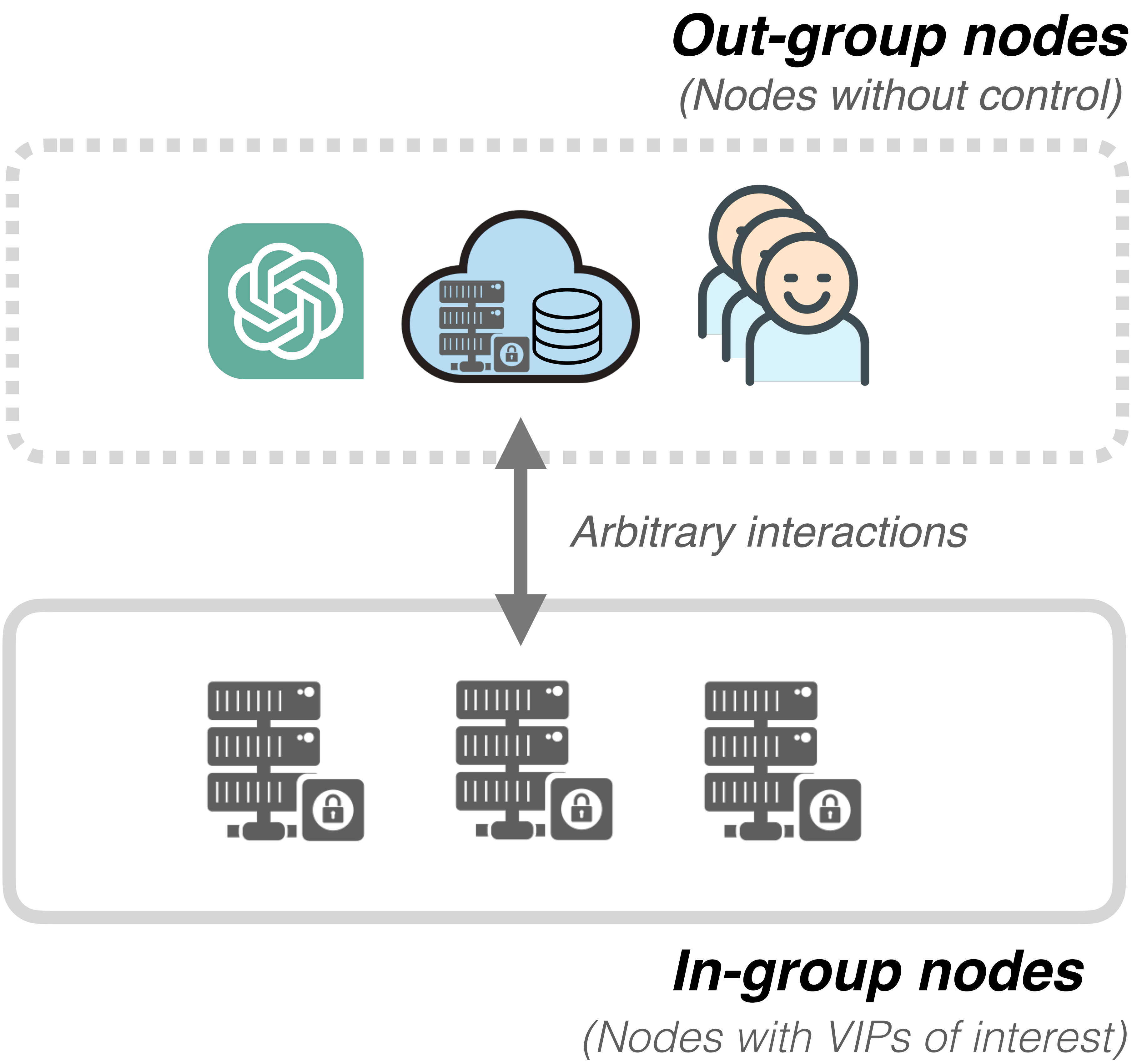
Revisiting classic snapshot protocols



Can we capture a *causally consistent* snapshot when a *subset* of the broader system participates?

A single external node can break the guarantee!

Beaver: practical partial snapshots



The same causal consistency abstraction

*Even when the target service interact with **external, black box** services (arbitrary number, scale, placement, or semantics) via **arbitrary pattern** (including multi-hop propagation of causal dependencies)*

Zero impact over existing service traffic

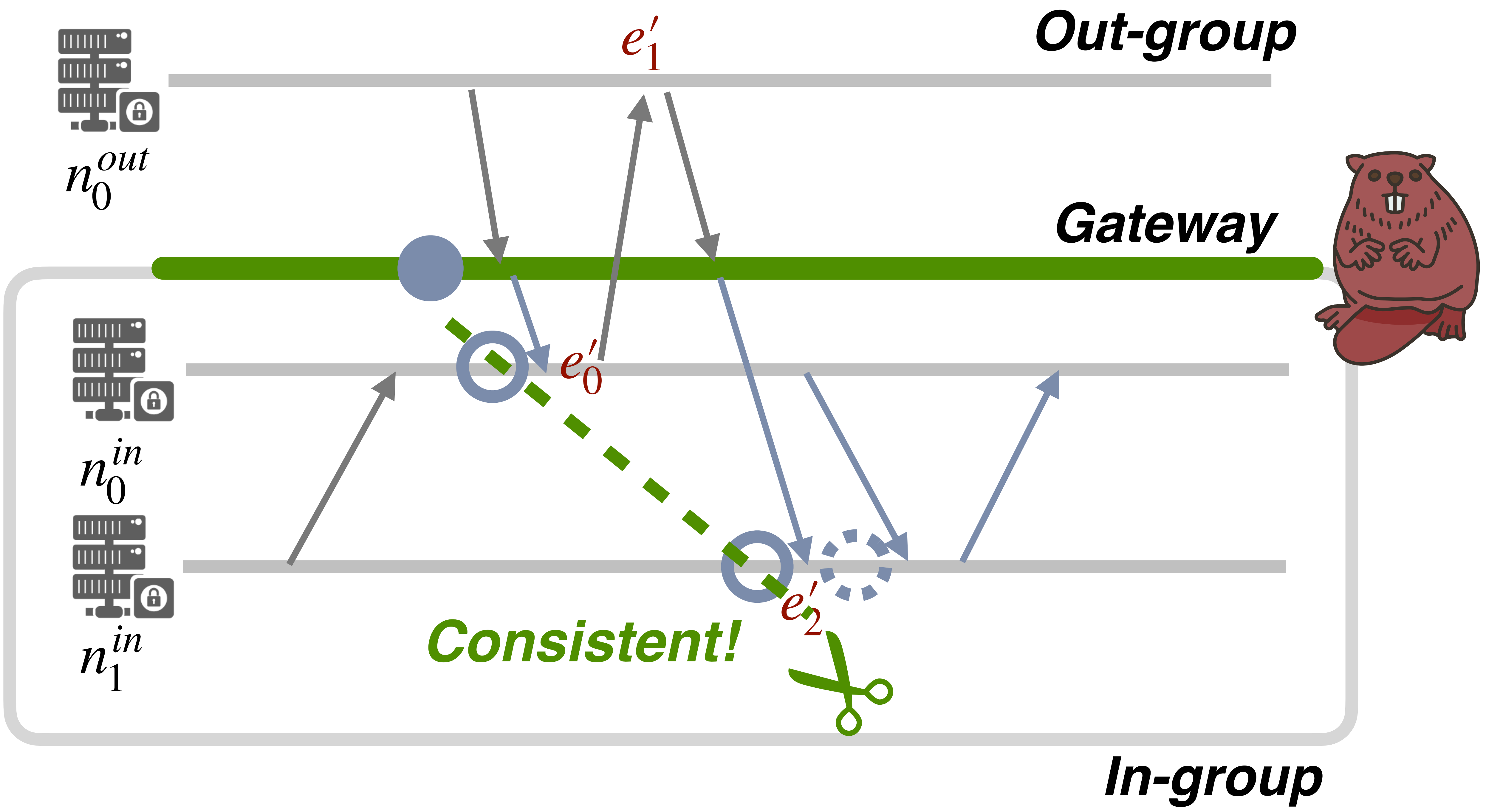
That is, absence of blocking or any form of delaying operations

[L. Yu et al., Beaver: Practical Partial Snapshots for Distributed Cloud Services, OSDI'24]

How is this possible without coordinating external machines?

Build a dam like a Beaver!

Gateway Marking



Beaver's gateway (GW) on a **load balancer**:

0. Handle all inbound traffic to the in-group
1. Initiate GW to enter snapshot out-of-band
2. Mark ***inbound*** packets correspondingly

The new cut ○ at n_1^{in} is before e'_2 (vs the previous cut ◌ which is after), consistent!

Gateway Marking

Theorem 1. *With MGM, a partial snapshot C_{part} for $P^{in} \subseteq P$ is causally consistent, that is, $\forall e \in C_{part}$, if $e'.p \in P^{in} \wedge e' \to e$, then $e' \in C_{part}$.*

Proof. Let $e.p = p_i^{in}$ and $e'.p = p_j^{in}$. There are 3 cases:

1. Both events occur in the same process, i.e., $i = j$.
2. $i \neq j$ and the causality relationship $e' \to e$ is imposed purely by in-group messages.
3. Otherwise, the causality relationship $e' \to e$ involves *at least* one $p \in P^{out}$.

In cases (1) and (2), the theorem is trivially true using identical logic to proofs of traditional distributed snapshot protocols. We prove (3) by contradiction.

Assume $(e \in C_{part}) \wedge (\exists e' \to e)$ but $(e' \notin C_{part})$. With (3), $e' \to e$ means that there must exist some e^{out} (at an out-group process) satisfying $e' \to e^{out} \to e$. Now, because $e' \notin C_{part}$, we know $e^{ss}_{p_j^{in}} \to e'$ or $e^{ss}_{p_j^{in}} = e'$, that is, p_j^{in}'s local snapshot happened before or during e'. Combined with the fact that the gateway is the original initiator of the snapshot protocol, we know that $e_g^{ss} \to e' \to e^{out} \to e$.

We can focus on a subset of the above causality chain: $e_g^{ss} \to e$. From the properties of the in-group snapshot protocol, $e_g^{ss} \to e$ implies that $e \notin C_{part}$.

This contradicts our original assumption that $e \in C_{part}$! □

Formal proof in paper

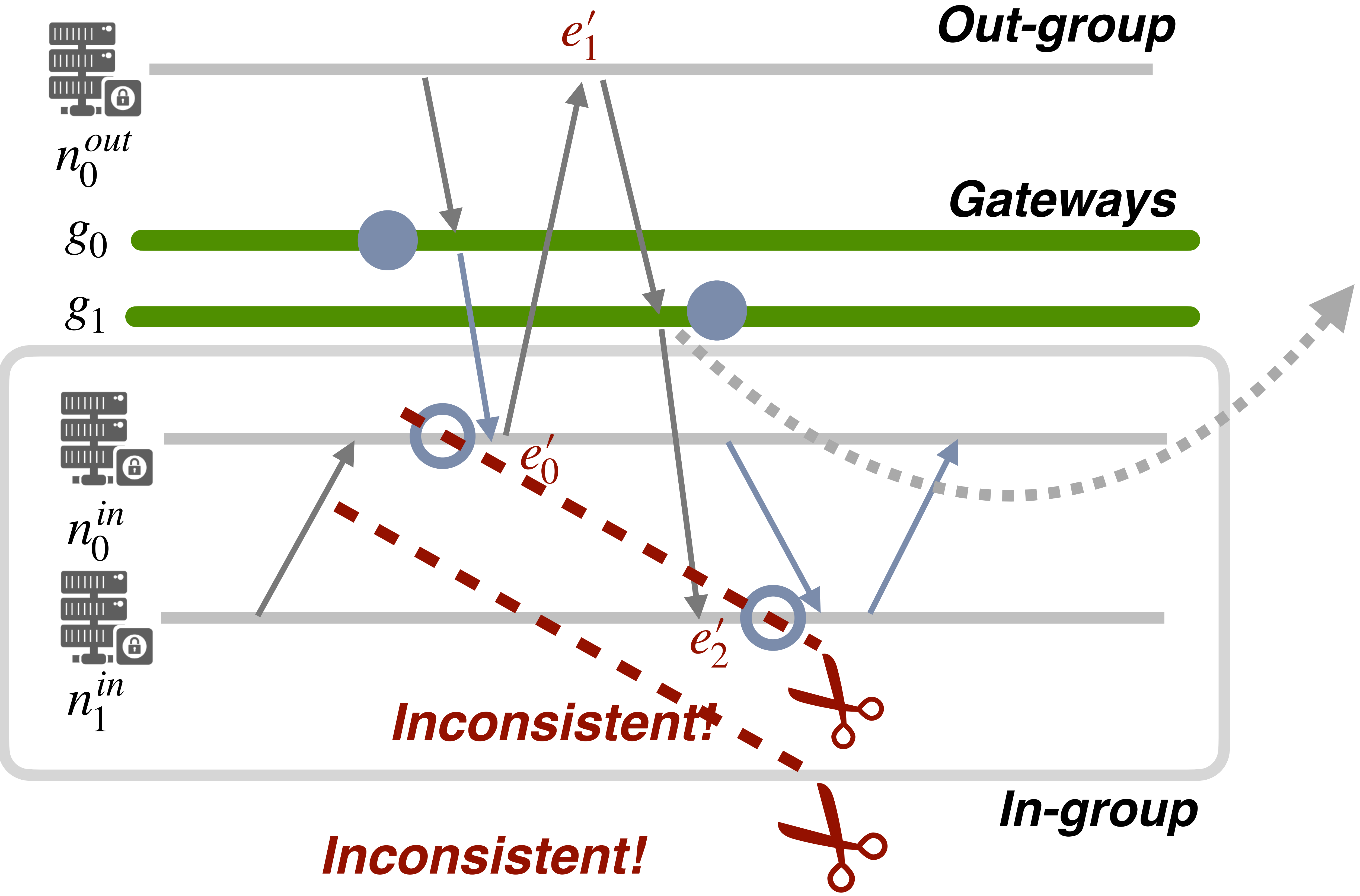
Holds even if treating the out-group nodes as black boxes

Sufficient to only observe the inbound messages

Challenge:

Real LB deployments aren’t monolithic: they have multiple gateway nodes

Challenge: Handling Multiple LBs



When message arrives, g_1 hasn't initiated the new snapshot mode to mark it, triggering the ***violation***

e'_2 in snapshot, yet e'_0 that leads to it is not, inconsistent!

Problem: initiating snapshot mode isn't atomic with multiple LBs

Optimistic Gateway Marking

Key idea: we don't *actually* need snapshot initiation to be **atomic**,
just to take less time than a round trip between in-group and out-group nodes

This is likely to be the case anyway!

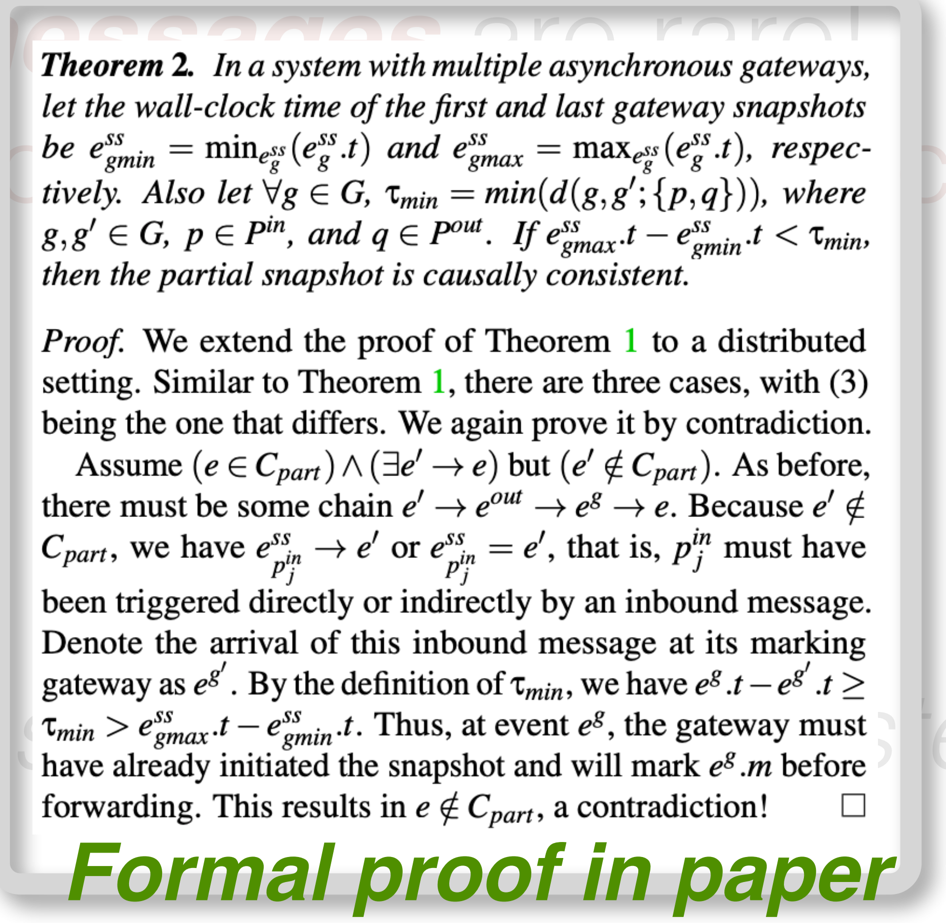
- A round trip between initiator and LB nodes (within the DC) is much faster than a RT between in-group and out-group nodes (outside the DC)

Optimistic approach: try taking a snapshot and reject it
if it takes too long to get response from all LB nodes

Correctness of Optimistic Gateway Marking

Theorem: if ◆ < ◆, the resulting snapshot is consistent!

- ◆ ≡ *Time gap between initiator-to-SLB one-way delays*
- ◆ ≡ *Time to form an external causal chain (GW→in-group→out-group→GW)*



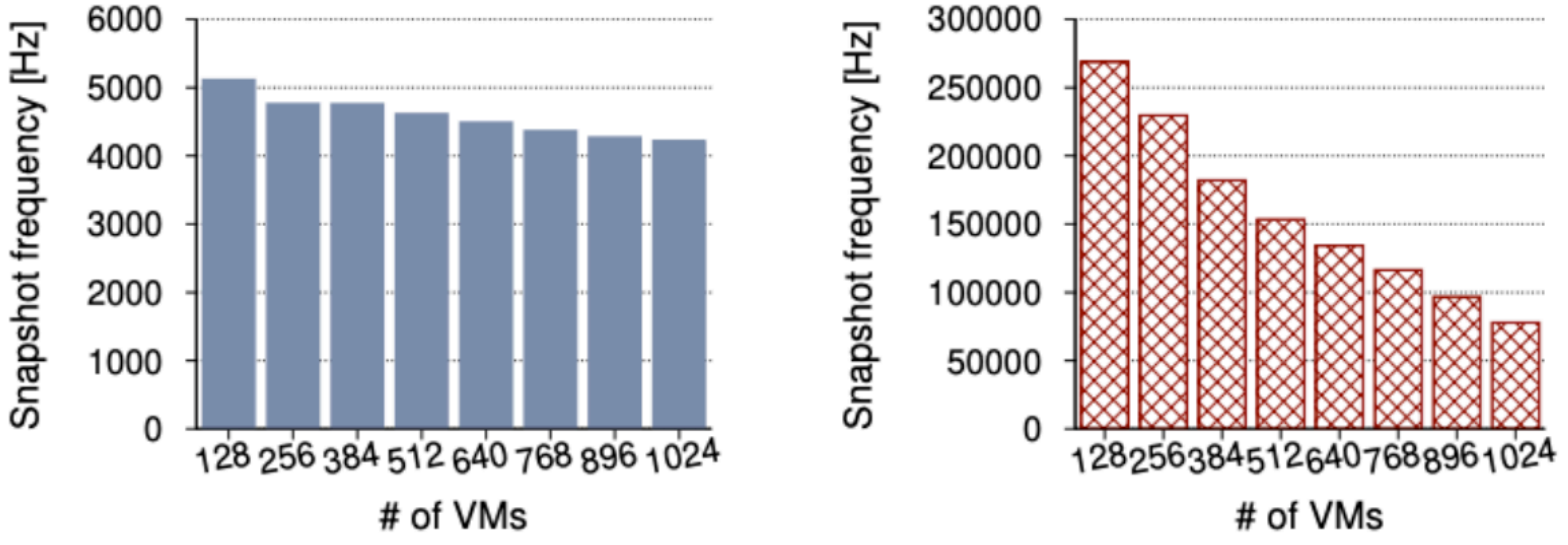
Theorem 2. *In a system with multiple asynchronous gateways, let the wall-clock time of the first and last gateway snapshots be $e^{ss}_{gmin} = \min_{e^{ss}_g}(e^{ss}_g.t)$ and $e^{ss}_{gmax} = \max_{e^{ss}_g}(e^{ss}_g.t)$, respectively. Also let $\forall g \in G$, $\tau_{min} = min(d(g, g'; \{p, q\}))$, where $g, g' \in G$, $p \in P^{in}$, and $q \in P^{out}$. If $e^{ss}_{gmax}.t - e^{ss}_{gmin}.t < \tau_{min}$, then the partial snapshot is causally consistent.*

Proof. We extend the proof of Theorem 1 to a distributed setting. Similar to Theorem 1, there are three cases, with (3) being the one that differs. We again prove it by contradiction.

Assume $(e \in C_{part}) \wedge (\exists e' \rightarrow e)$ but $(e' \notin C_{part})$. As before, there must be some chain $e' \rightarrow e^{out} \rightarrow e^g \rightarrow e$. Because $e' \notin C_{part}$, we have $e^{ss}_{p^{in}_j} \rightarrow e'$ or $e^{ss}_{p^{in}_j} = e'$, that is, p^{in}_j must have been triggered directly or indirectly by an inbound message. Denote the arrival of this inbound message at its marking gateway as $e^{g'}$. By the definition of τ_{min}, we have $e^g.t - e^{g'}.t \geq \tau_{min} > e^{ss}_{gmax}.t - e^{ss}_{gmin}.t$. Thus, at event e^g, the gateway must have already initiated the snapshot and will mark $e^g.m$ before forwarding. This results in $e \notin C_{part}$, a contradiction! □

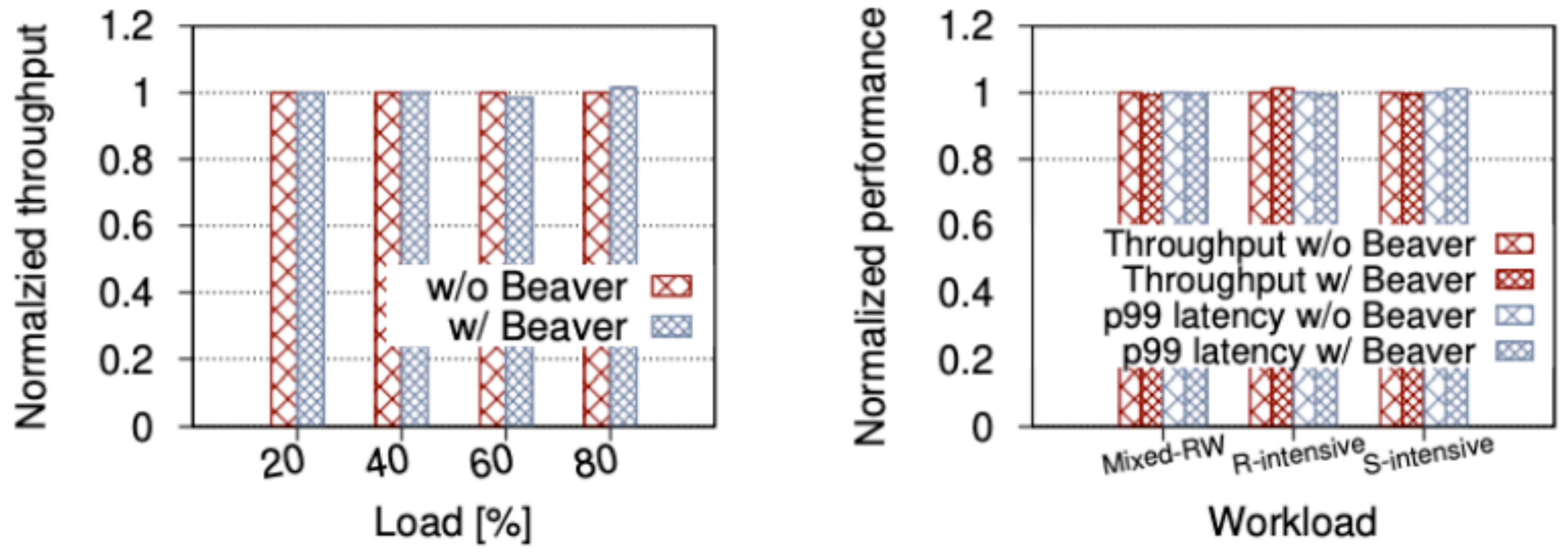
Formal proof in paper

Beaver supports fast snapshots without performance impact



(a) w/o parallelism (b) w/ parallelism

Beaver supports fast snapshot rates



(a) Stressed workloads (b) YCSB benchmarks

Beaver incurs zero performance impact

Beaver summary

First protocol to extend classic consistent snapshot protocols to
practical cloud settings

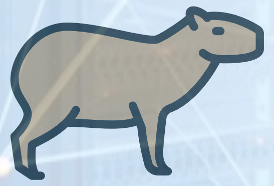
Ensures **causal consistency** with minimal changes and minimal overhead

Key approach: integrate simple functionality to support snapshots
into flexible, HW-accelerated load balancer

Finale

Accelerated cloud-scale load balance is important for efficiency
and also provides opportunities for new features

Distributed systems can take advantage of these

-  Moving data to transparently handle skewed workloads
-  Transparently migrating active connections between servers
-  Checkpointing systems without instrumenting all participants

Cloud infrastructure brings **distributed systems and networking** together in a powerful new way!