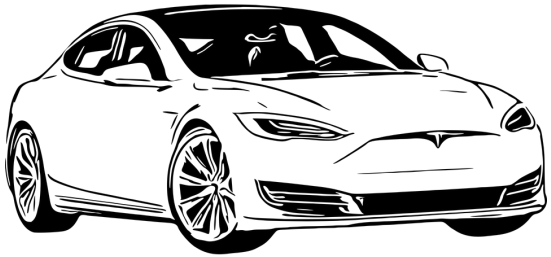
Reconfiguration and Extensibility For Low-Latency Key-Value Stores

Chinmay Kulkarni
University of Utah

DOORDASH



Millions of Devices are Generating Millions of Events Every Second

Sensors, Devices, IoT...

Events, Data

Cloud Services

Aggregated in the Cloud to Gain Insights and Drive Application Logic

Sensors,
Devices, IoT...

Events,
Data

Cloud
Services

Ad hoc
Analysis

Streaming

Ingest

Query

State Management

State Management

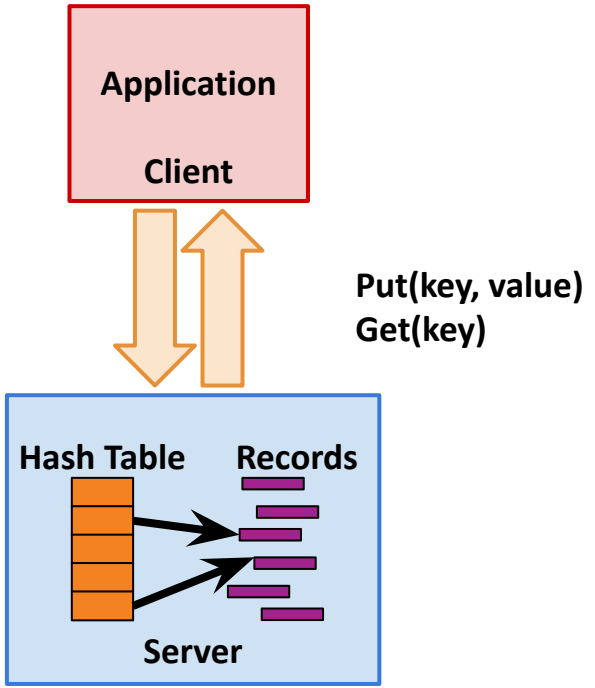
The crucial piece in the pipeline

- ➔ Focal point for massive number of events and queries
- ➔ Responsible for ingestion, indexing, availability, fault-tolerance etc.

Typically a Key-Value data model (Key-Value Store)

- ➔ **key(**User-id**)**: **value{** Linkedin profile **}**
- ➔ **key(**Pacemaker Device-id**)**: **value{** # of heartbeats in the last n minutes **}**

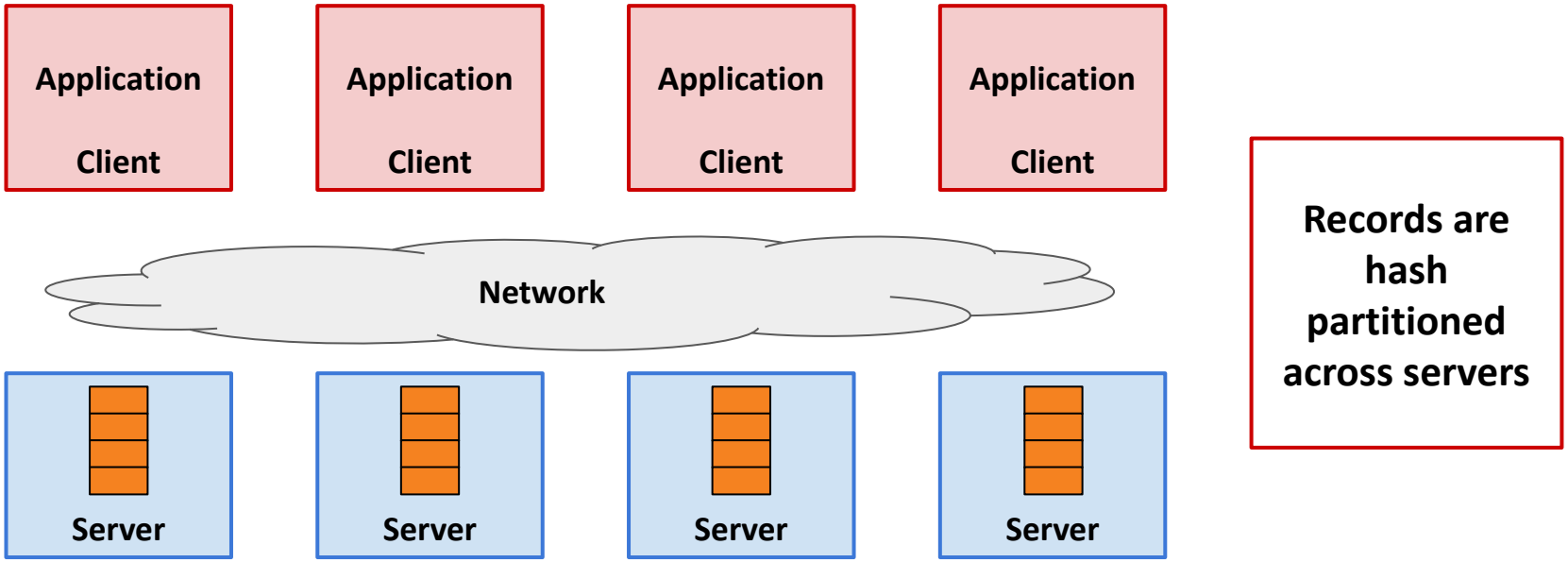
Key-Value Stores: Data Model



Records belong to a 64 bit hash space

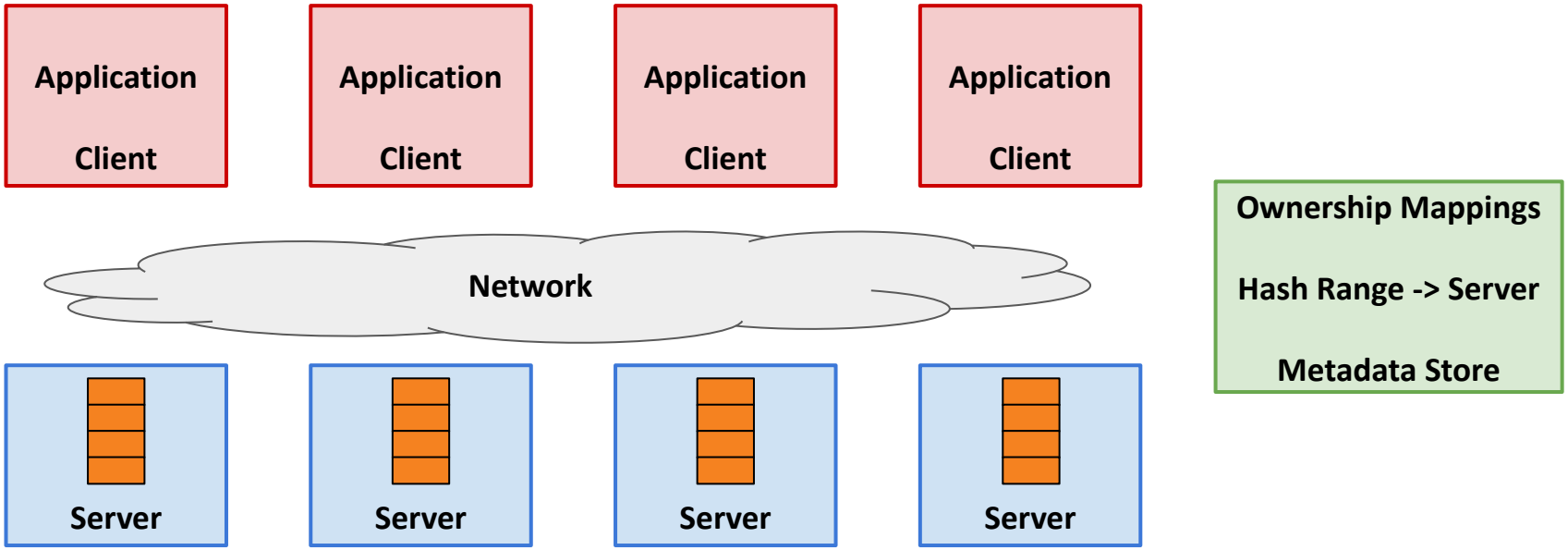
$[0, 2^{63} - 1)$

Key-Value Stores: Typical Deployment



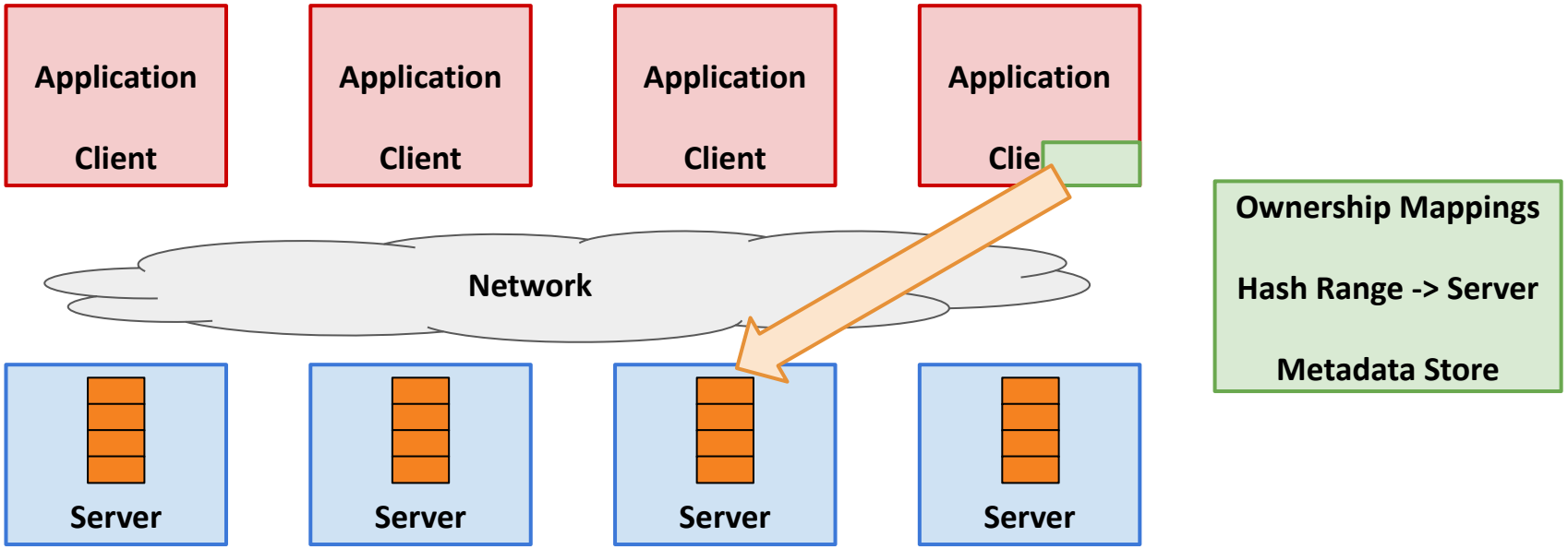
Disaggregate Servers from Clients over Network, Scale-out each Independently

Key-Value Stores: Record Ownership



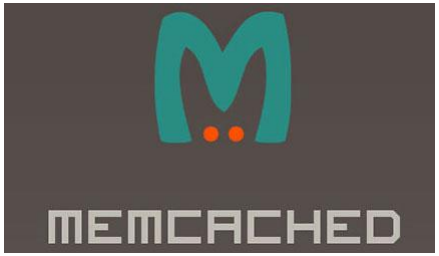
Servers own partitions: order requests, provide durability and crash recovery

Key-Value Stores: Record Ownership

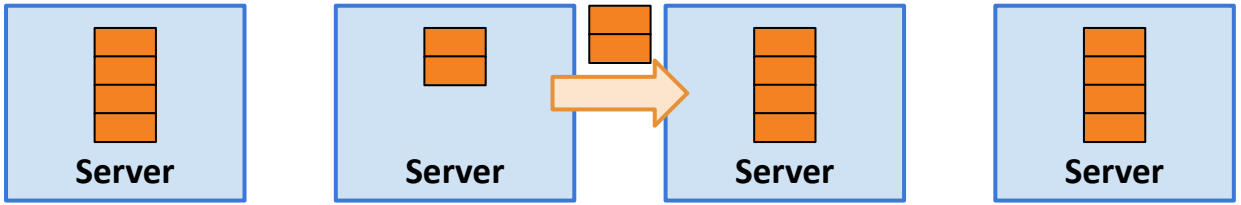


Clients cache mappings, route requests to correct owner

Key-Value Stores: Example Systems



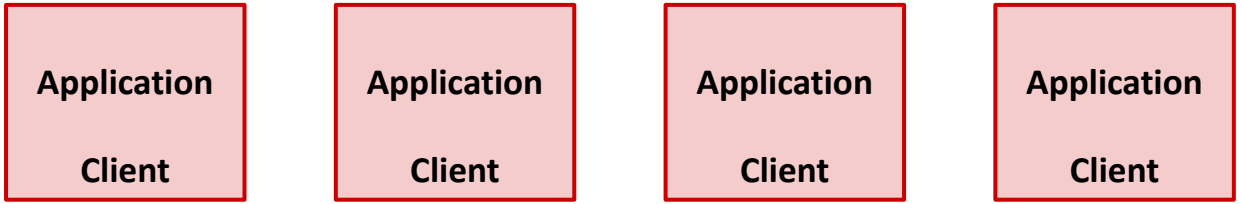
Key-Value Stores: Data Migration



Servers often need to horizontally re-distribute load and data (Slicer, OSDI'16)

- ➔ Clusters need to be scaled up and scaled down
- ➔ Workloads change; access patterns, load imbalances etc
- ➔ "Planned failures"; server maintenance, software updates etc

Key-Value Stores: Extensibility



Put(key, value) , Get(key), assoc_range(key), assoc_get(key)

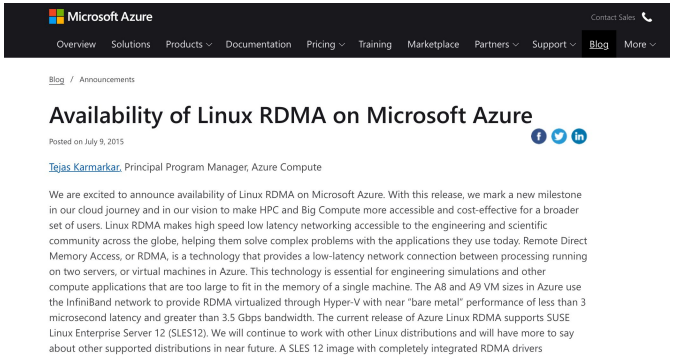
Some applications need rich data model/vertical interface (Facebook Tao, ATC'13)

- ➔ Get friend list (assoc_get())
- ➔ Get friends of friends (assoc_range())

New Wave of Key-Value Stores

Use Kernel-Bypass to leverage High Speed Networks

- ➔ Hardware accelerated transport. Ex: Mellanox Infiniband, Project Catapult, SmartNICs etc.
- ➔ Transport integrated into application. Ex: DPDK, RDMA etc.
- ➔ MICA(NSDI'14) , FaRM (NSDI'14, SOSP'15), KVDirect (SOSP'17) etc



Low-Latency (10s of microseconds per request), High throughput (Million reqs/s)

New Wave of (Impractical) Key-Value Stores

Stripped down design to maximize normal case performance:

- ➔ Reconfiguration? Rich Data models? Multi-tenancy?
- ➔ Supporting the above can cut into normal-case performance. Ex: Record ownership checks

Fast key-value stores:
An idea whose time has come and gone

Atul Adya, Robert Grandl, Daniel Myers
Google

Henry Qin
Stanford University

Abstract

Remote, in-memory key-value (RINK) stores such as Memcached [6] and Redis [7] are widely used in industry and are an active area of academic research. Coupled with stateless application servers to execute business logic and a database-like system to provide persistent storage, they form a core component of popular data center service architectures. We

First, they may provide a cache over a storage system to enable faster retrieval of persistent state. Second, they may store short-lived data, such as per-session state, that does not warrant persistence [25].

These stores enable services to be deployed using *stateless application servers* [16], which maintain only per-request state; all other state resides in persistent storage or in a

HotOS 2019

My thesis: Mechanisms for reconfiguration, rich data models and multi-tenancy without compromising on performance

Thesis Contributions

Fast Data Migration with Low Latency Impact [Rocksteady, SOSP'17]:

➔ Enables quick reconfiguration in response to load imbalances, changes in access patterns etc

Low-cost Coordination for Requests Dispatch and Migration [Shadowfax, VLDB'21(Revision)]:

➔ Enables 8x higher throughput on cloud VMs without stalls and cross-core coordination

Extensibility and Multi-tenancy without Hardware Isolation [Splinter, OSDI'18]:

➔ Enables diverse data models by allowing tenants to push functions to the store at runtime

Thesis Statement

Low-latency stores adopt stripped down designs that optimize for the normal case, trading off features that would make them more practical.

My thesis shows that this trade off is unnecessary.

Carefully leveraging and extending abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their performance.

Fast Data Migration

Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. **Rocksteady: Fast Migration for Low-latency In-memory Storage.** In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*, **SOSP'17**

Chinmay Kulkarni, Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. **Beyond Simple Request Processing with RAMCloud.** *IEEE Data Engineering Bulletin*, **IEEE DEB 40(1) (March 2017)**

Rocksteady Introduction

Distributed low-latency in-memory key-value stores are emerging

- ➔ Predictable response times ~**10 µs median, ~60 µs 99.9th-tile**

Problem: Must migrate data between servers

- ➔ Quickly respond to hot spots, skew shifts, load spikes
- ➔ Minimize performance impact of migration

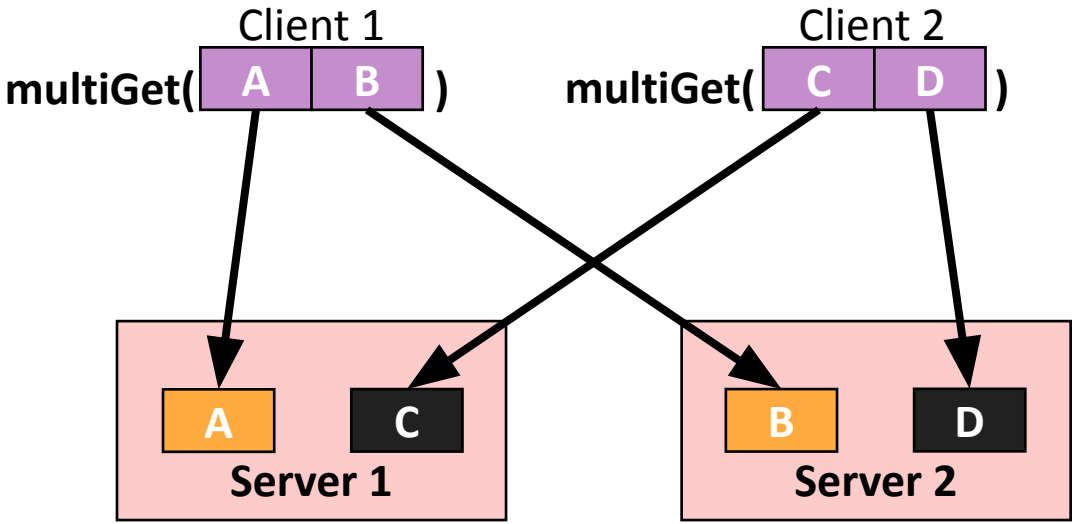
Solution: **Fast** data migration with **low latency impact**

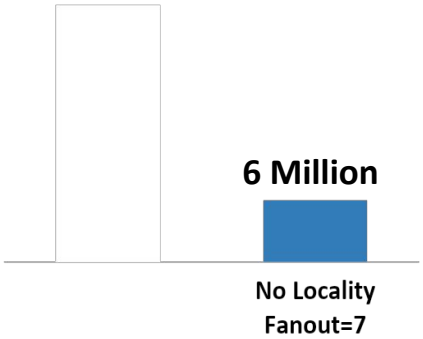
- ➔ Early ownership transfer of data, leverage workload skew
- ➔ Low priority, parallel and adaptive migration

Result: Migration protocol for RAMCloud in-memory key-value store

- ➔ Migrates **6x** faster than the baseline

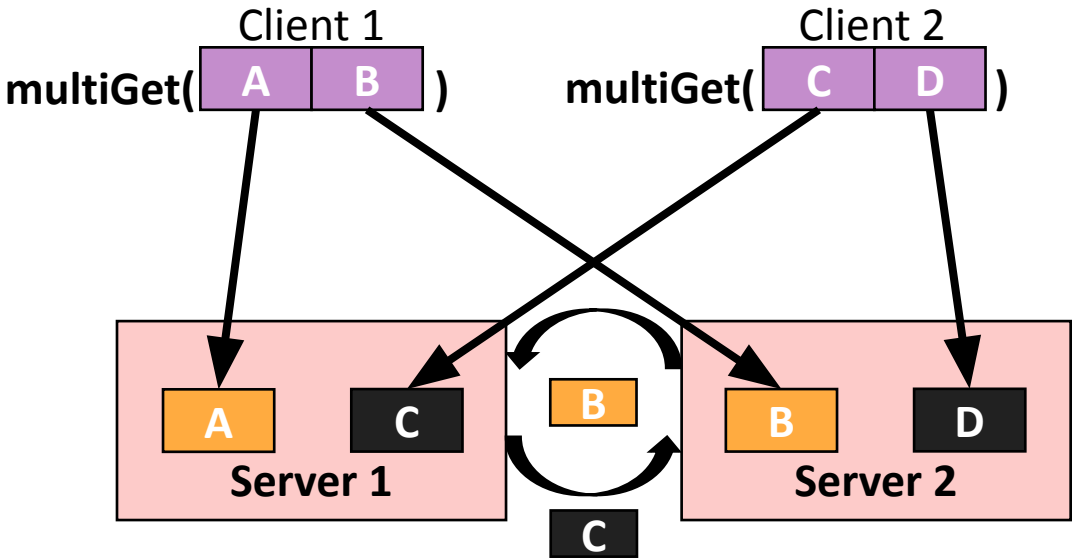
Why Migrate Data?

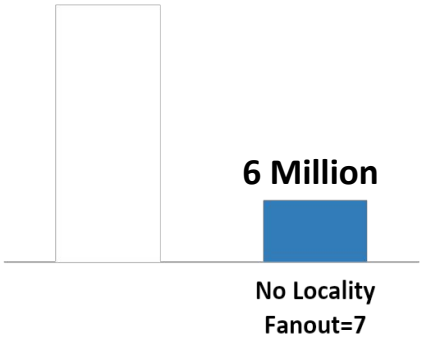




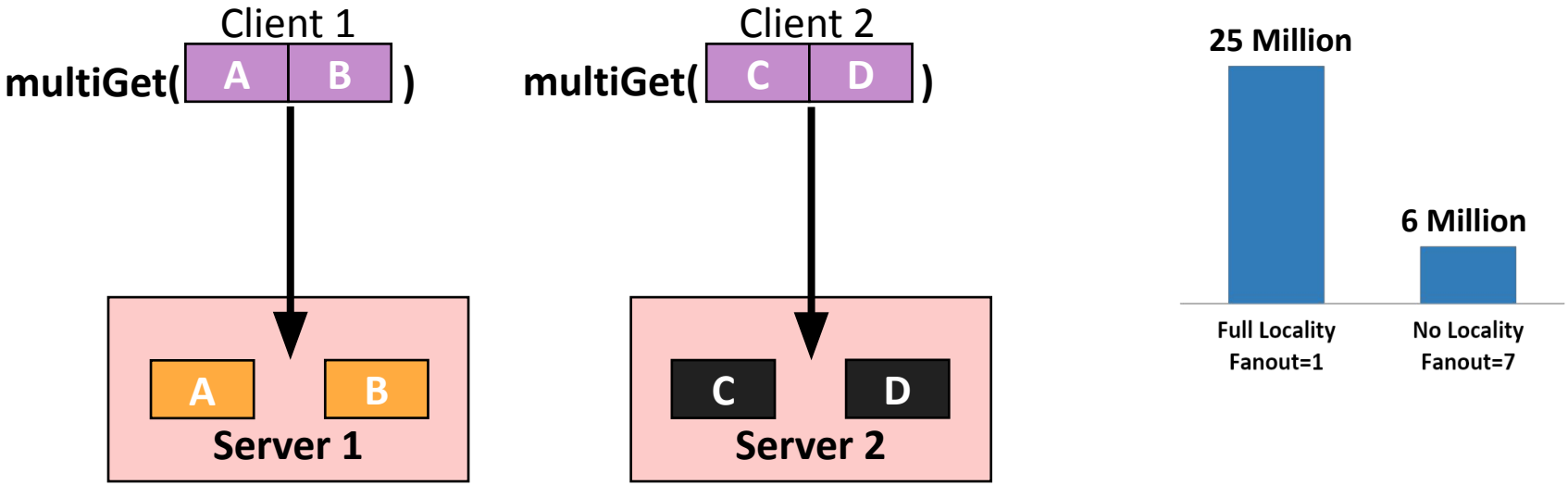
Poor spatial locality → High multiGet() fan-out → More RPCs

Migrate To Improve Spatial Locality





Spatial Locality Improves Throughput



Better spatial locality → Fewer RPCs → Higher throughput

Benefits multiGet(), range scans

Performance Goals For Migration

Migrate data fast

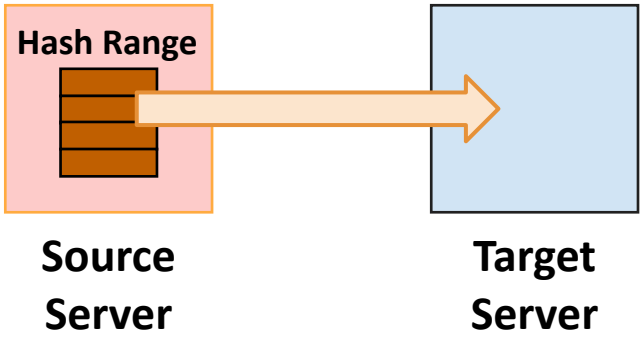
➔ Workloads dynamic ⟶ Respond quickly

Maintain low access latency

➔ 10 µsec median latency ⟶ System extremely sensitive

➔ Tail latency matters at scale ⟶ Even more sensitive

Rocksteady: Design



Instantaneous ownership transfer

- ➔ Immediate load reduction at overloaded source
- ➔ Creates “headroom” for migration work

Leverage skew to rapidly migrate hot data

- ➔ Most workloads are skewed, Target comes up to speed with little data movement

Adaptive parallel, pipelined at source and target

- ➔ All cores avoid stalls, but yield to client-facing operations

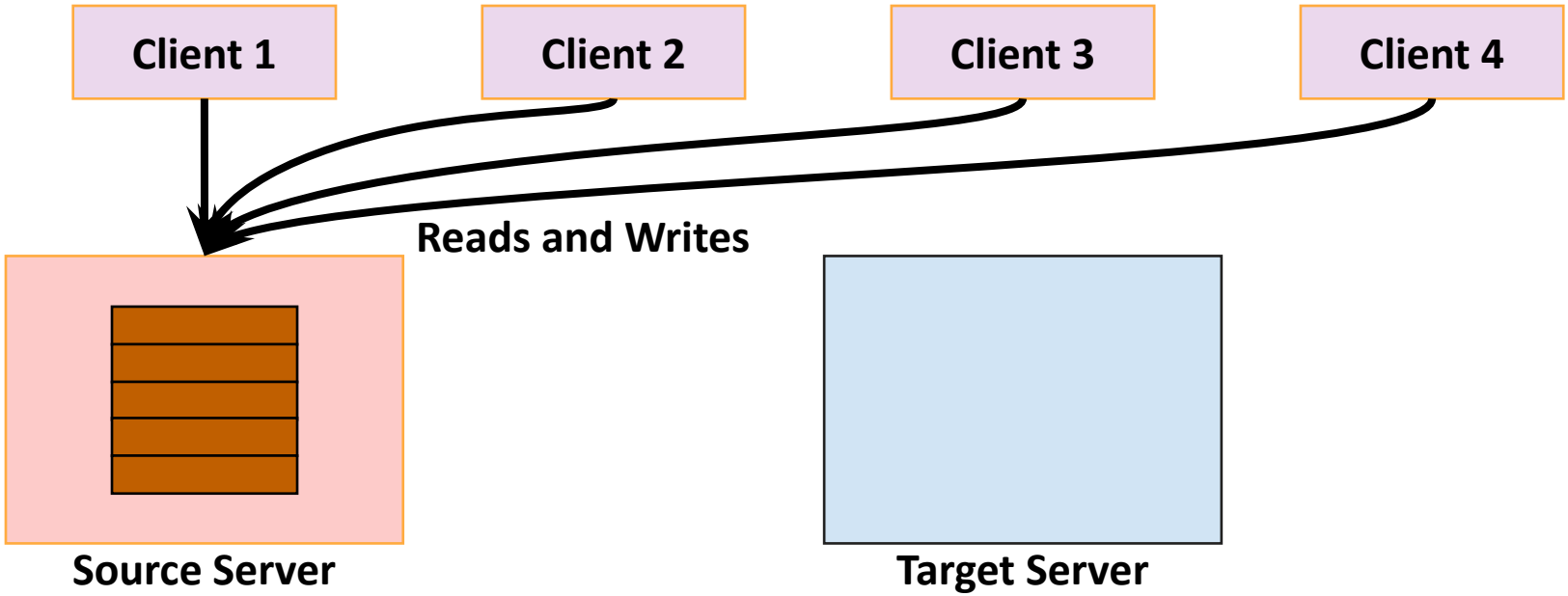
Safely defer replication at target

- ➔ Eliminates replication bottleneck and contention

Rocksteady: Early Ownership Transfer

Problem: Loaded source can bottleneck migration

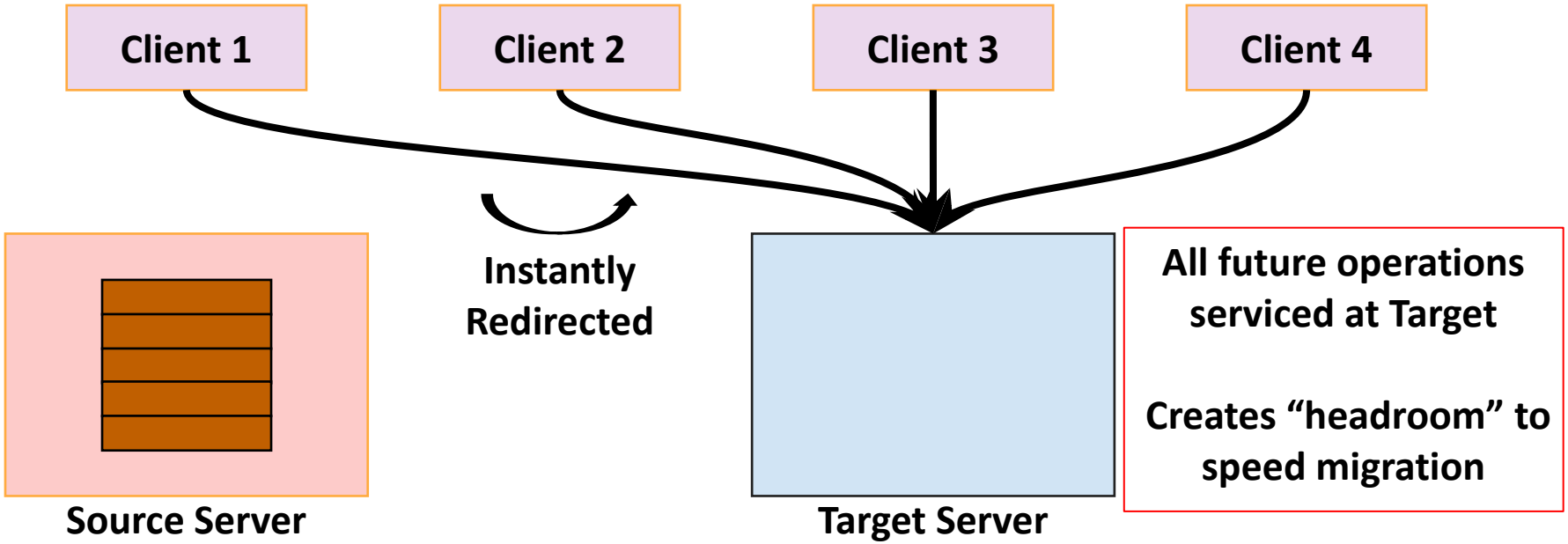
Solution: Instantly shift ownership and all load to target before shifting data



Rocksteady: Early Ownership Transfer

Problem: Loaded source can bottleneck migration

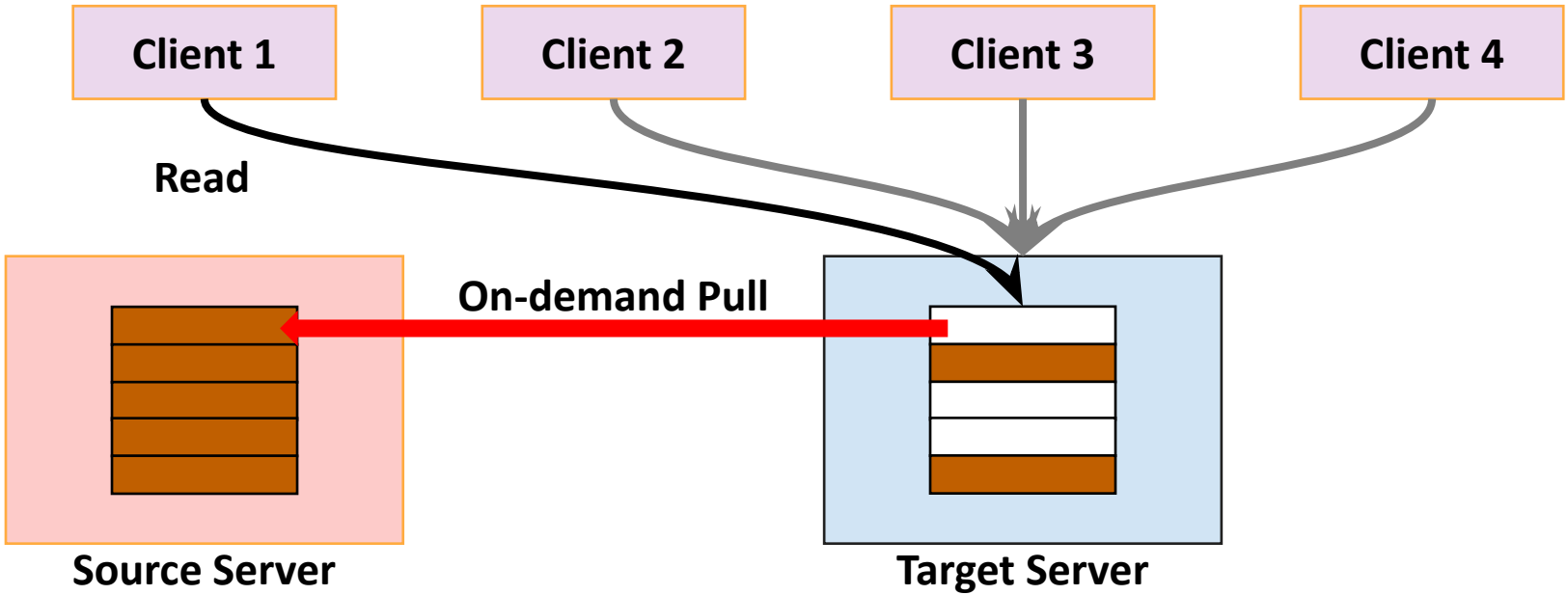
Solution: Instantly shift ownership and all load to target before shifting data



Rocksteady: Leverage Skew

Problem: Data has not arrived at source yet

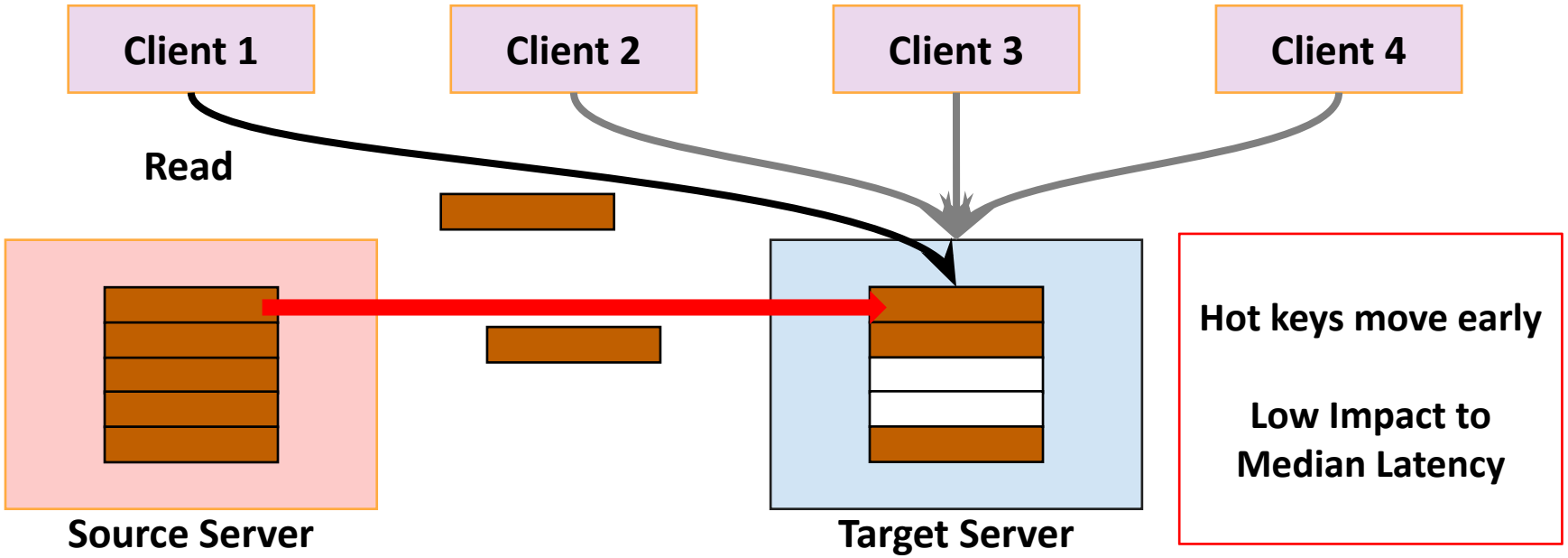
Solution: On demand migration of unavailable data



Rocksteady: Leverage Skew

Problem: Data has not arrived at source yet

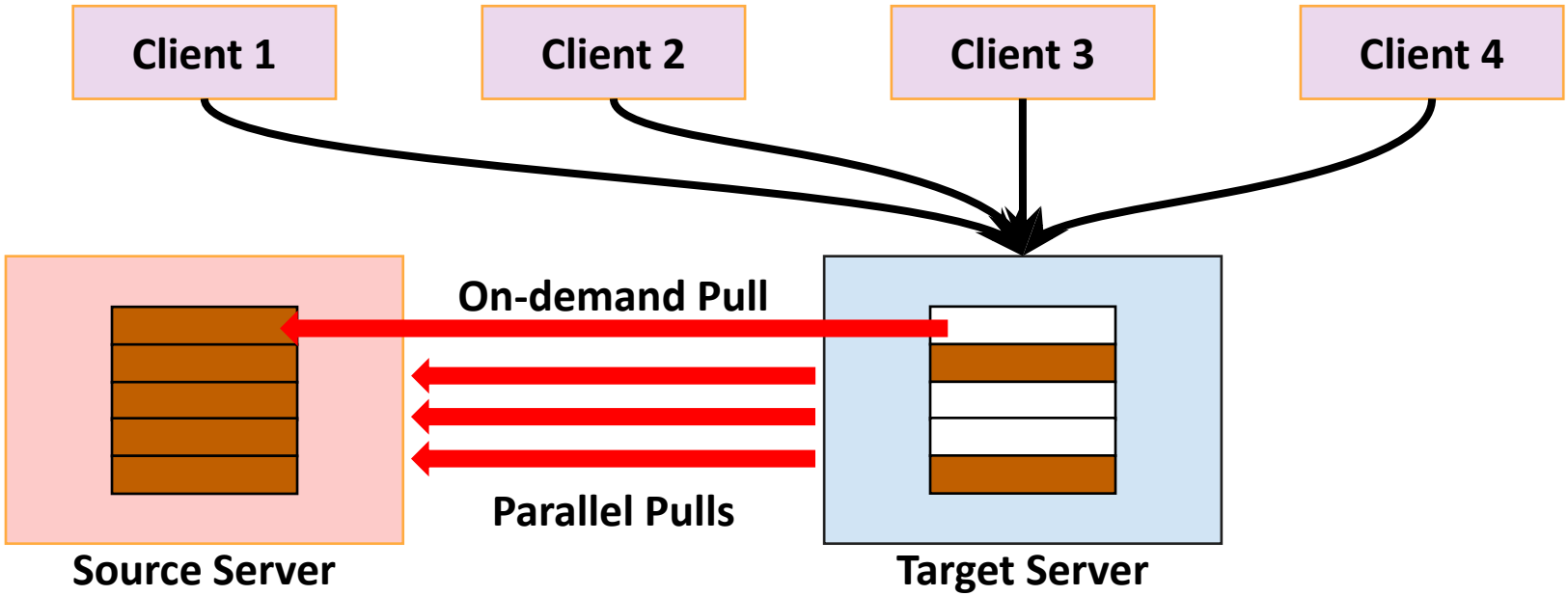
Solution: On demand migration of unavailable data



Rocksteady: Adaptive and Parallel

Problem: Old single-threaded protocol limited to 130 MB/s

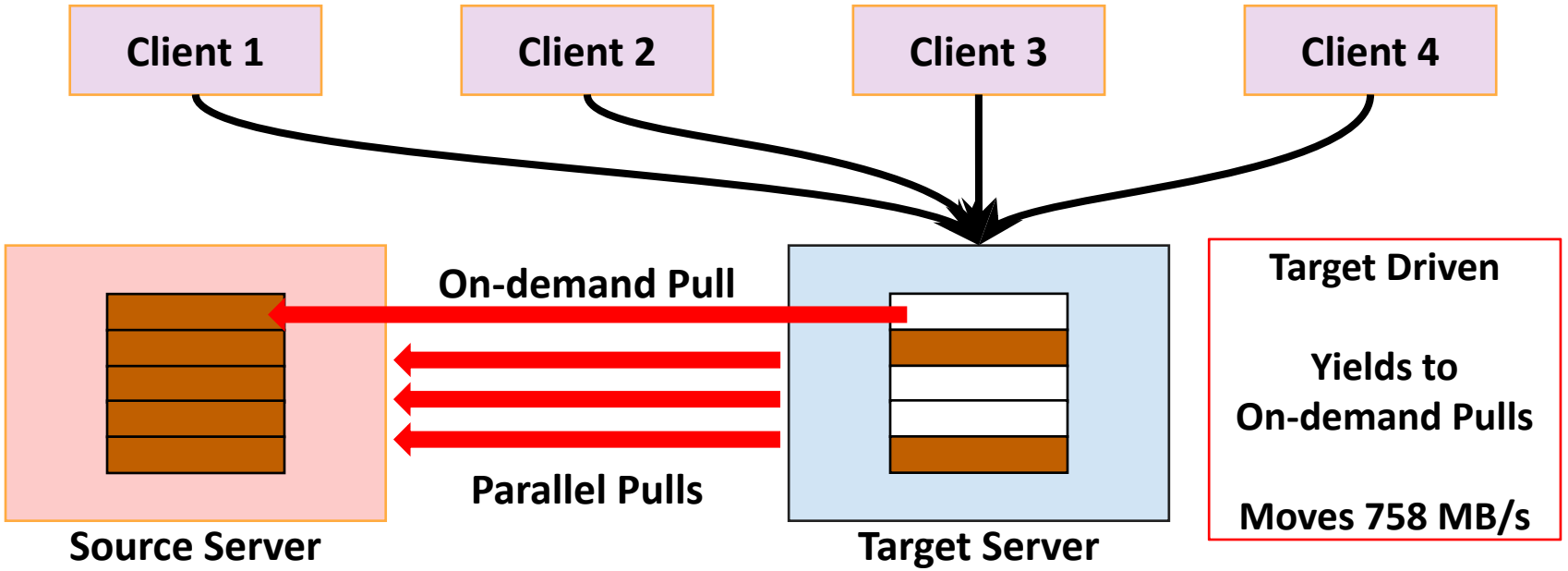
Solution: Pipelined and parallel at source and target



Rocksteady: Adaptive and Parallel

Problem: Old single-threaded protocol limited to 130 MB/s

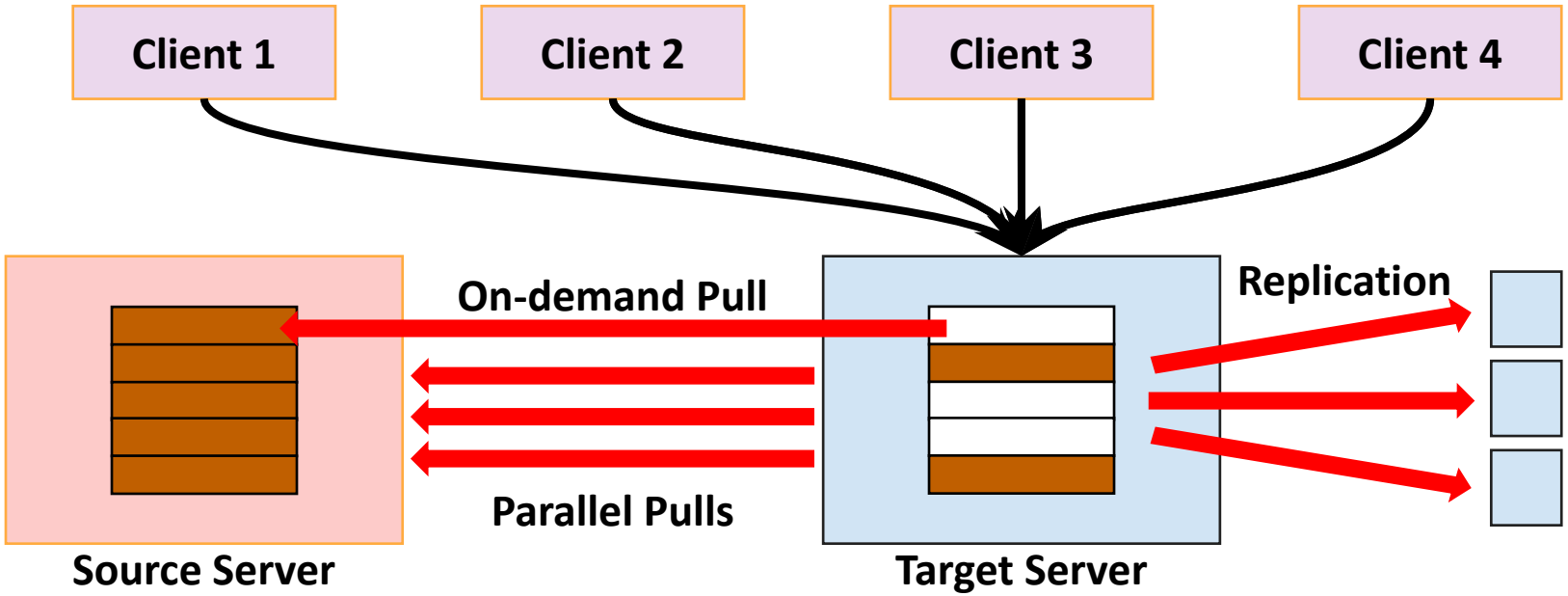
Solution: Pipelined and parallel at source and target



Rocksteady: Eliminate Sync Replication

Problem: Synchronous replication bottleneck at target

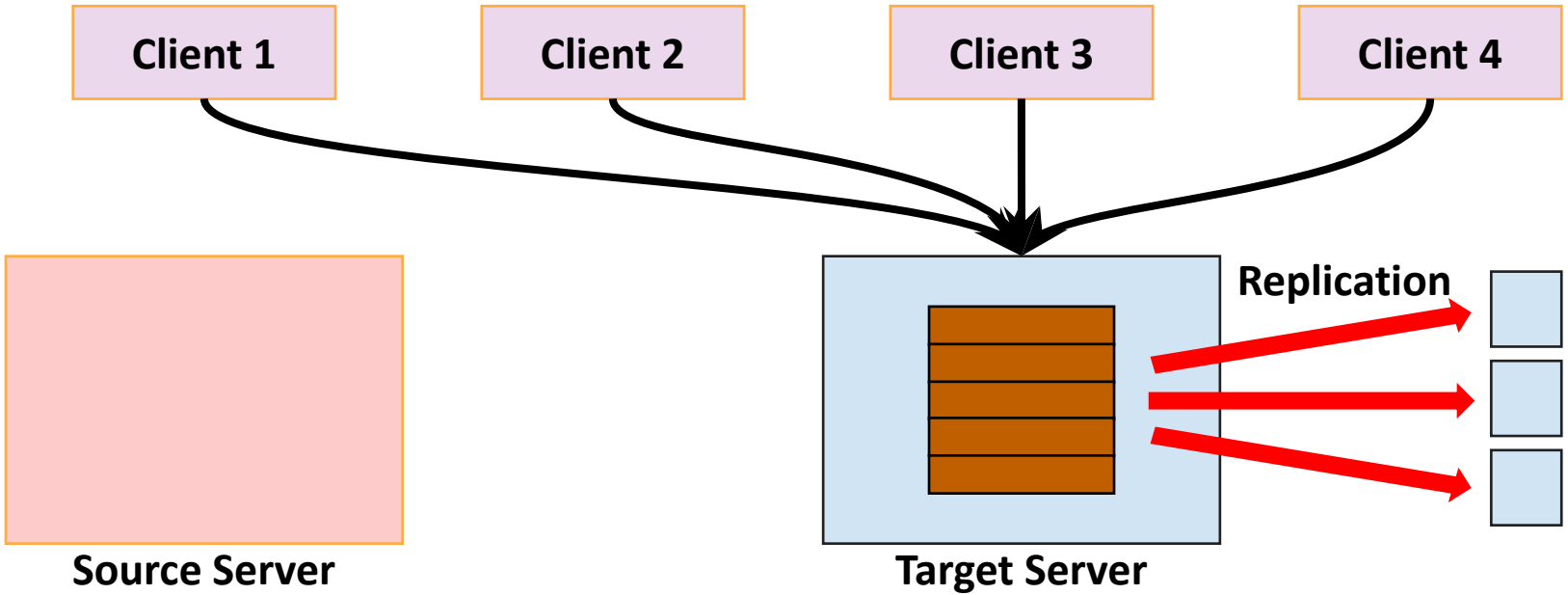
Solution: Safely defer replication until after migration



Rocksteady: Eliminate Sync Replication

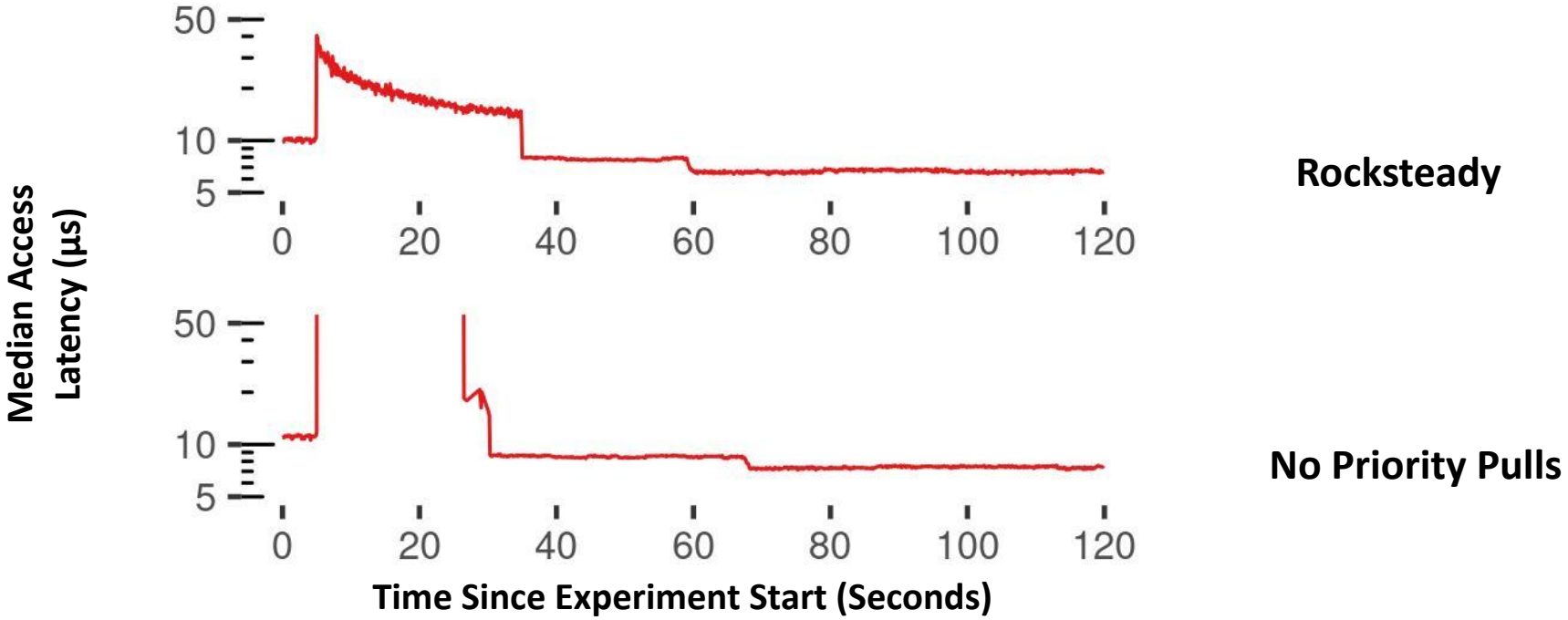
Problem: Synchronous replication bottleneck at target

Solution: Safely defer replication until after migration



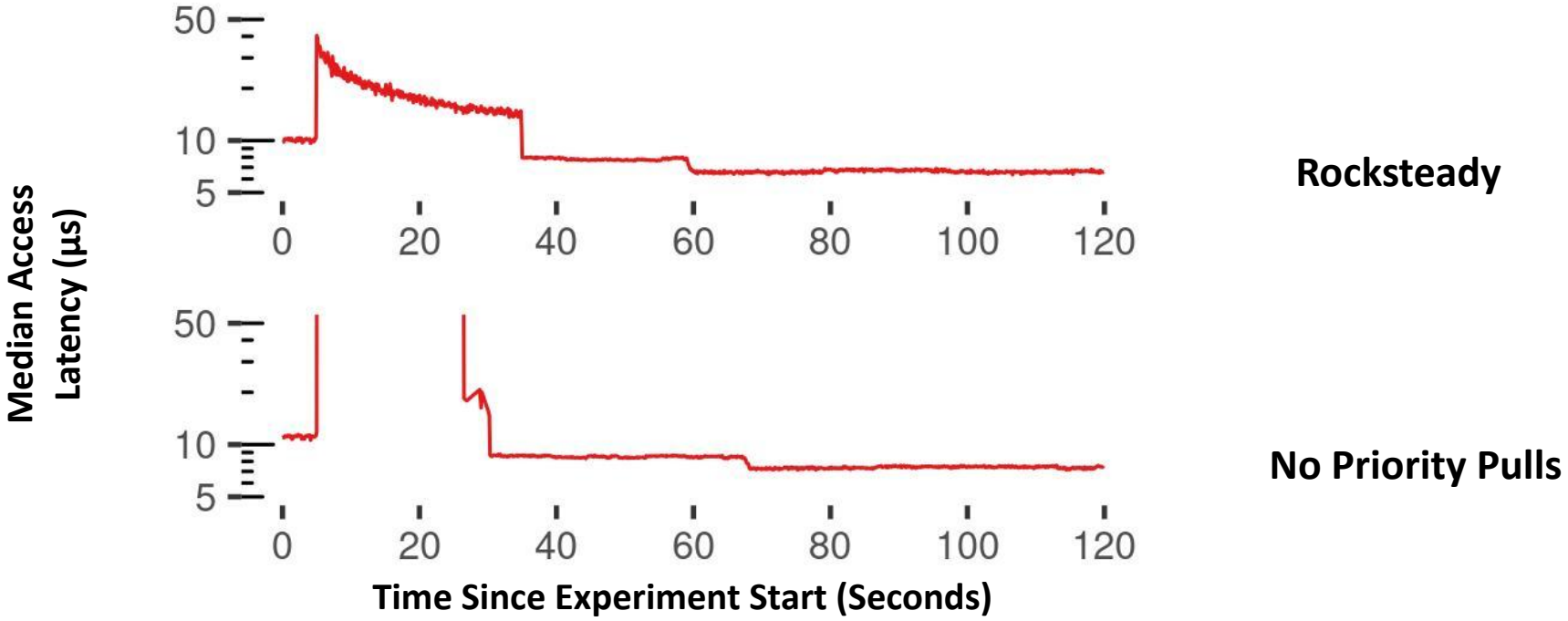
Performance of Rocksteady

YCSB-B, 300 Million objects (30 B key, 100 B value), migrate half



Performance of Rocksteady

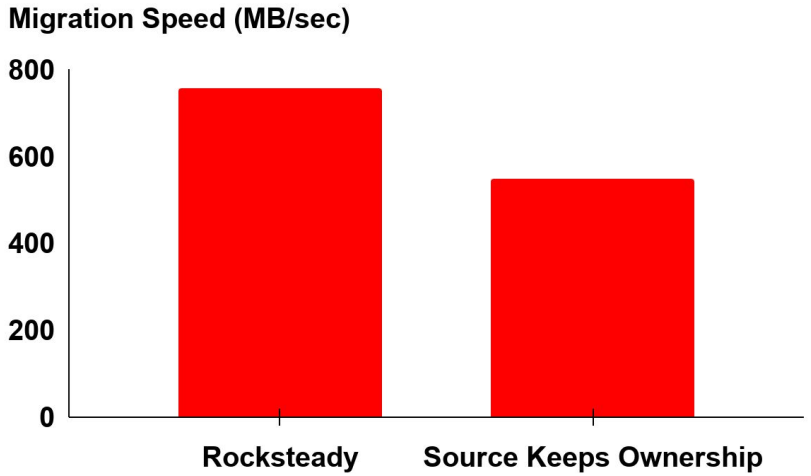
YCSB-B, 300 Million objects (30 B key, 100 B value), migrate half



Priority Pulls help maintain Median Latency

Performance of Rocksteady

YCSB-B, 300 Million objects (30 B key, 100 B value), migrate half



Source headroom speeds up migration by 28%

Rocksteady Conclusion

Distributed low-latency in-memory key-value stores are emerging

- ➔ Predictable response times ~**10 µs median, ~60 µs 99.9th-tile**

Problem: Must migrate data between servers

- ➔ Quickly respond to hot spots, skew shifts, load spikes
- ➔ Minimize performance impact of migration

Solution: **Fast** data migration with **low latency impact**

- ➔ Early ownership transfer of data, leverage workload skew
- ➔ Low priority, parallel and adaptive migration

Result: Migration protocol for RAMCloud in-memory key-value store

- ➔ Migrates **28%** faster than the baseline

Low-Cost Coordination

Chinmay Kulkarni, Badrish Chandramouli, and Ryan Stutsman. **Achieving High Throughput and Elasticity in a Larger-than-Memory Store. Under Revision** at **VLDB 2021**

Shadowfax Introduction

Multi-core optimized single-node key-value stores are emerging

- ➔ **Very** high throughput ~**100 Million events/sec/server.** Ex: FASTER (SIGMOD'18)

Problem: Retain high throughput in the public cloud

- ➔ **Avoid request dispatch and network bottlenecks** → Saturate servers within hash index
- ➔ Support reconfiguration/migration while preserving throughput

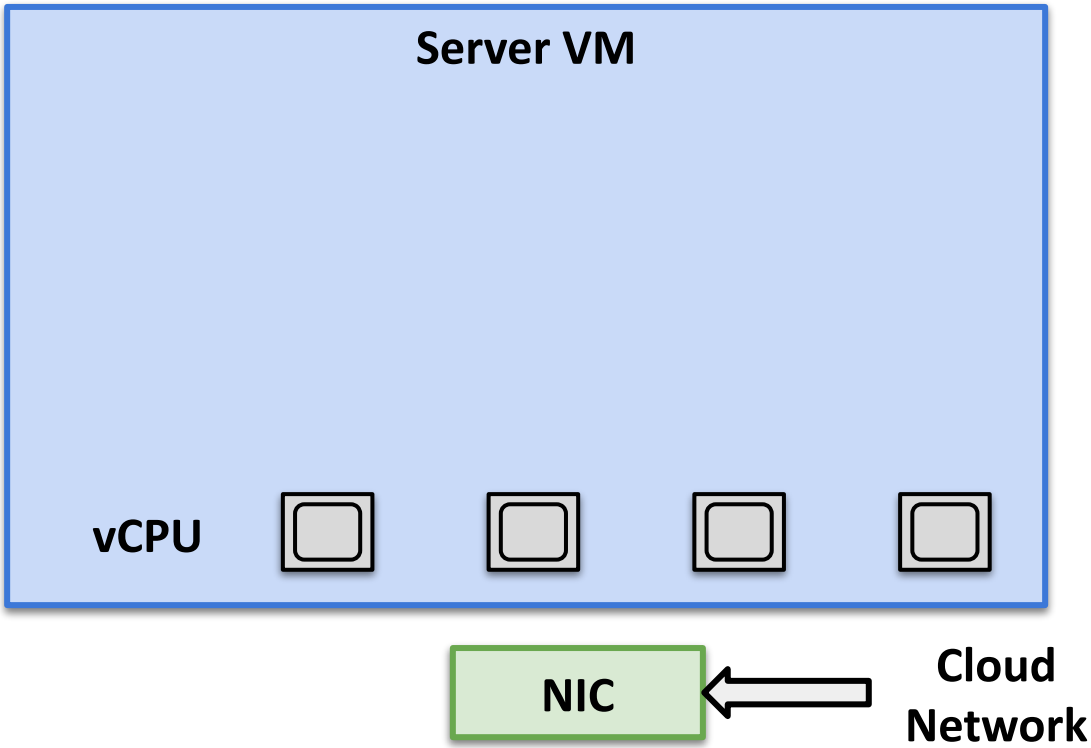
Solution: Partitioned Request Dispatch, Asynchronous Global Cuts

- ➔ Cores share FASTER index, request dispatch layer is partitioned → **retain high throughput**
- ➔ Cores avoid coordination during migration → **maintain high throughput**

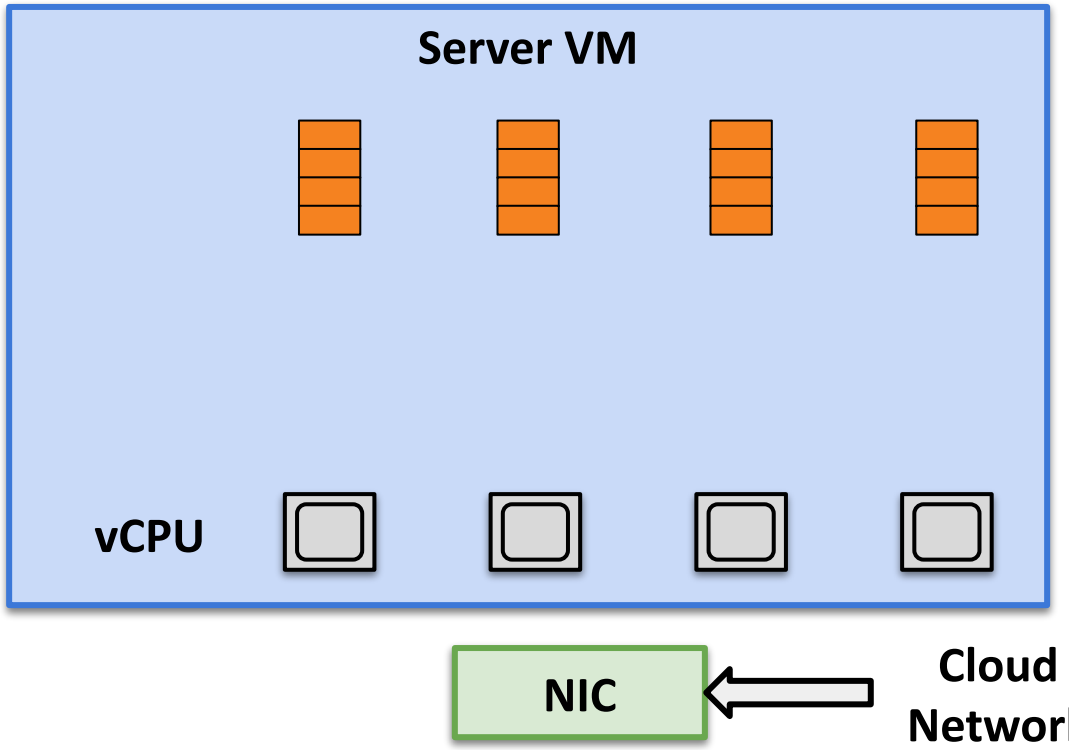
Shadowfax: High Throughput, Elastic, Larger-than-Memory key-value store

- ➔ **100 Million events/sec/VM on Azure**, **80 Million events/sec/VM** during migration
- ➔ Spans DRAM, SSD and Cloud blob storage

Seastar: State-of-the-art Cloud Key-Value Store



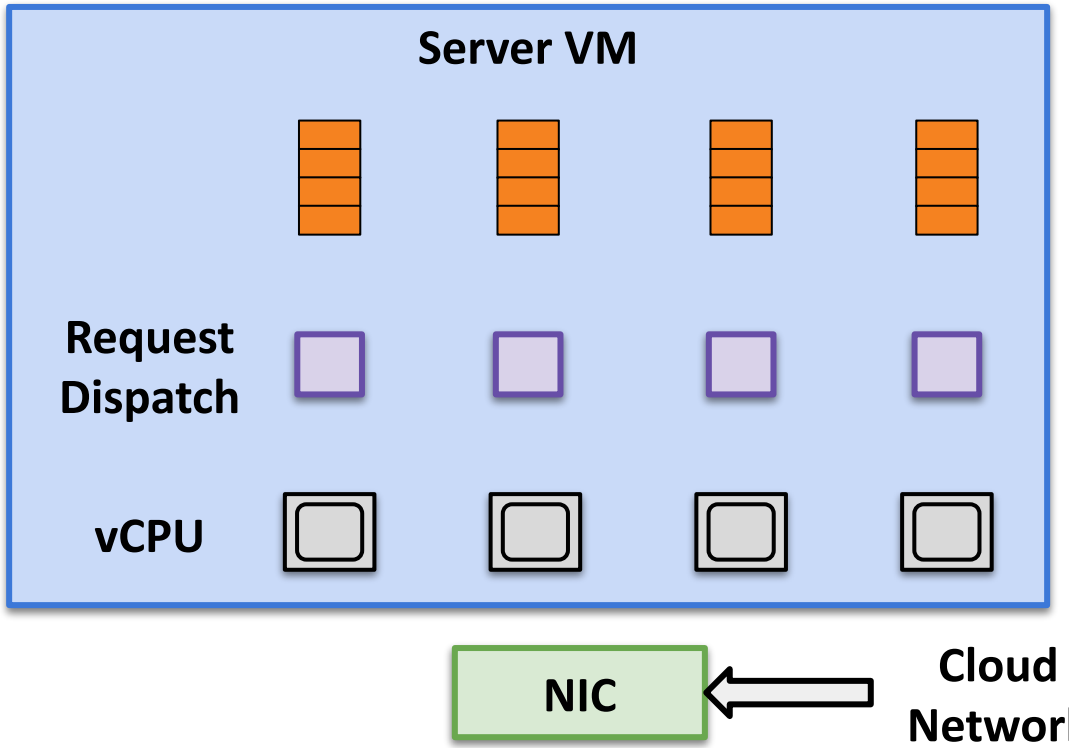
Seastar: Records Partitioned Across Cores



Hash partition records across vCPUs

Avoids synchronization on records

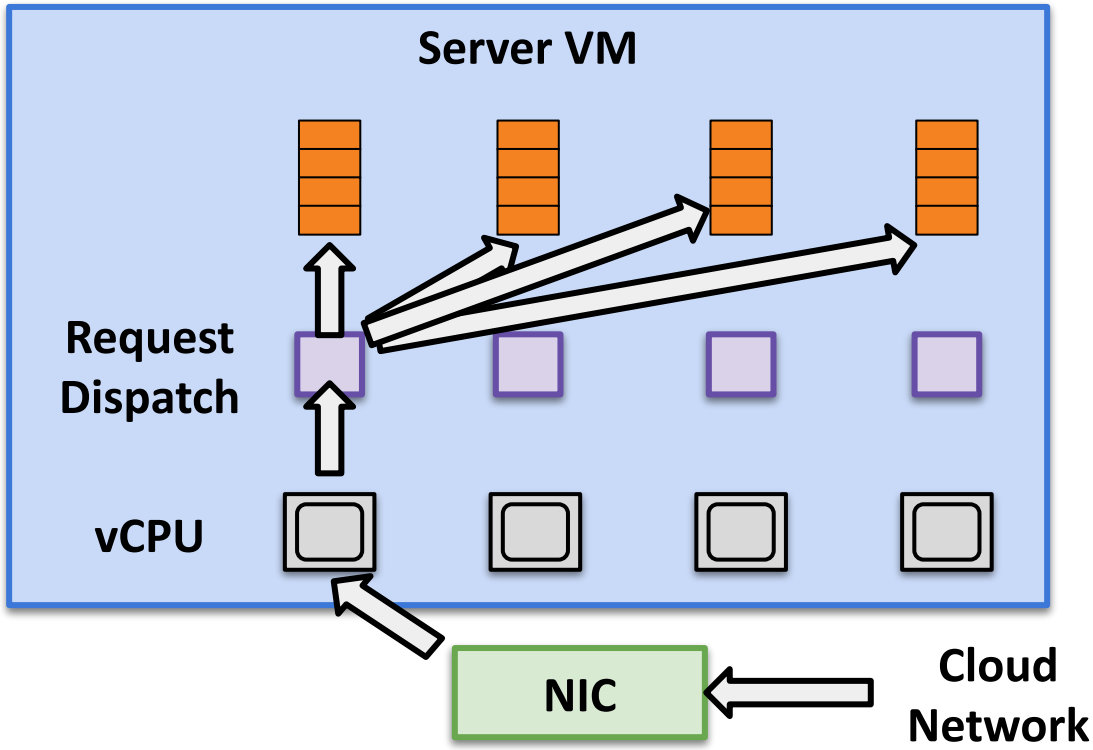
Seastar: Records Partitioned Across Cores



Dispatcher on every vCPU

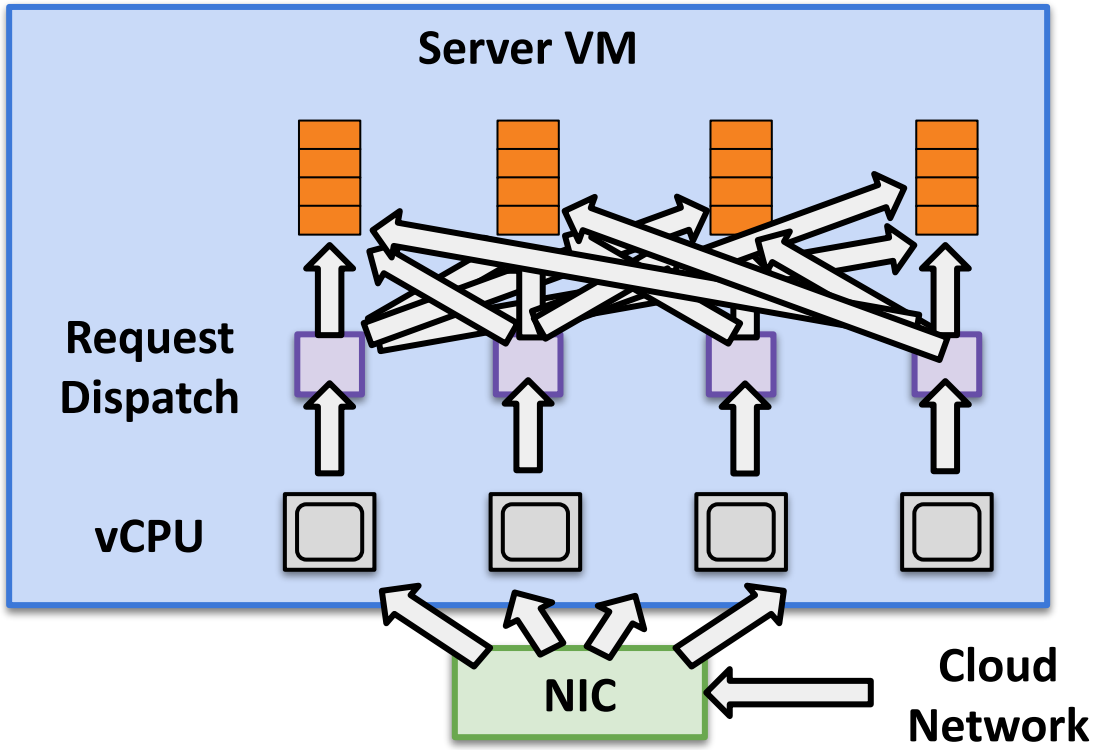
Each listens on different TCP port

Seastar: Requests Routed Within Server



Each dispatcher routes requests to correct partition

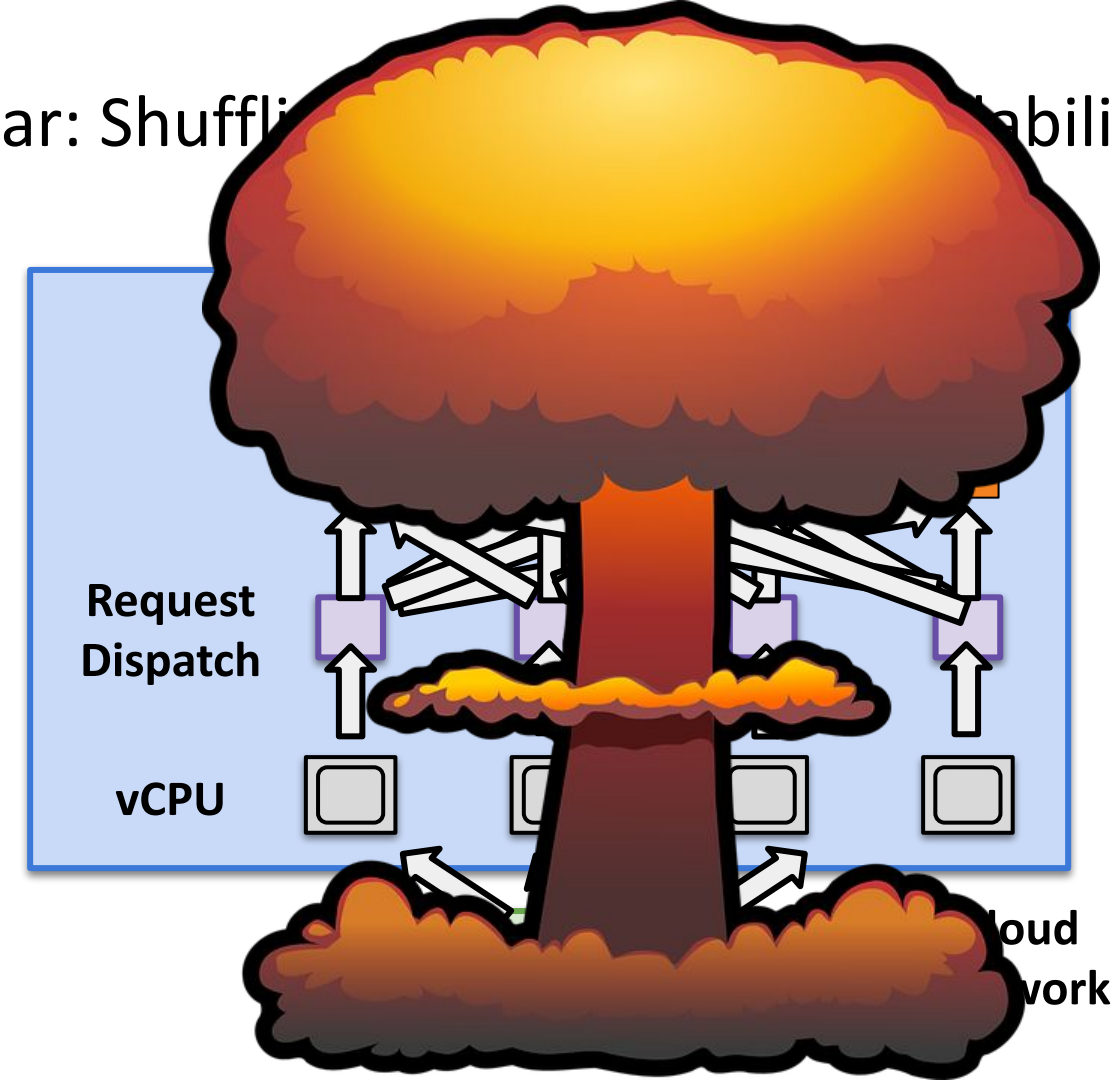
Seastar: Shuffling Requests Limits Scalability



Each dispatcher routes requests to correct partition

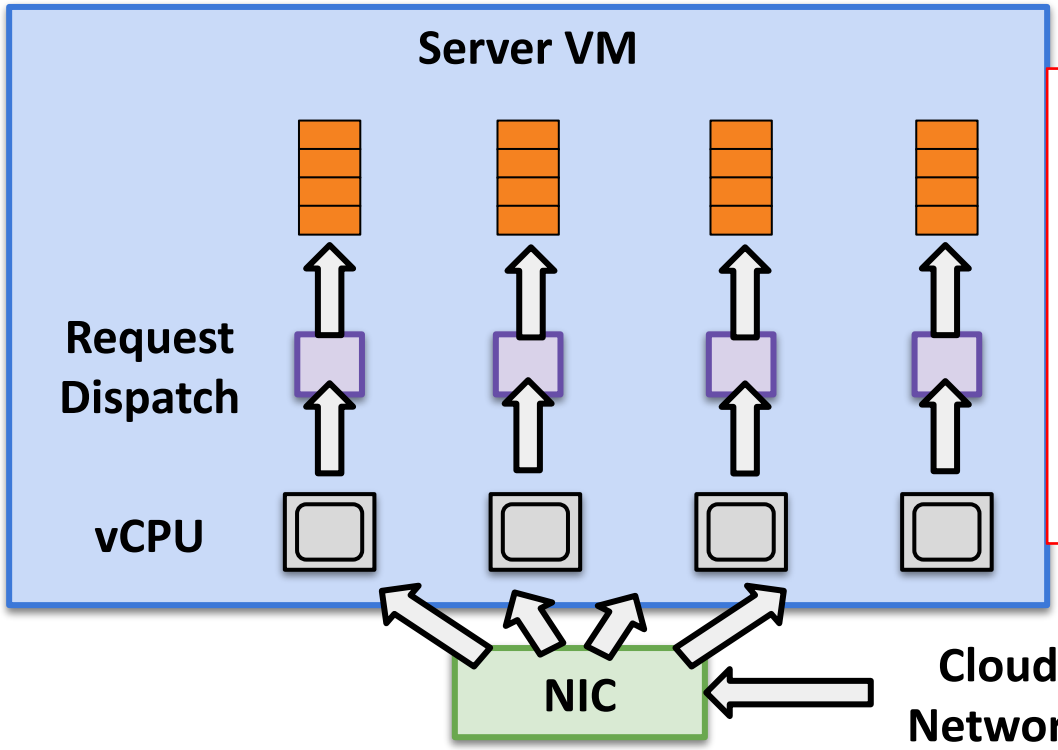
Expensive shuffle!

Seastar: Shuffli ability



Doesn’t scale across vCPUs

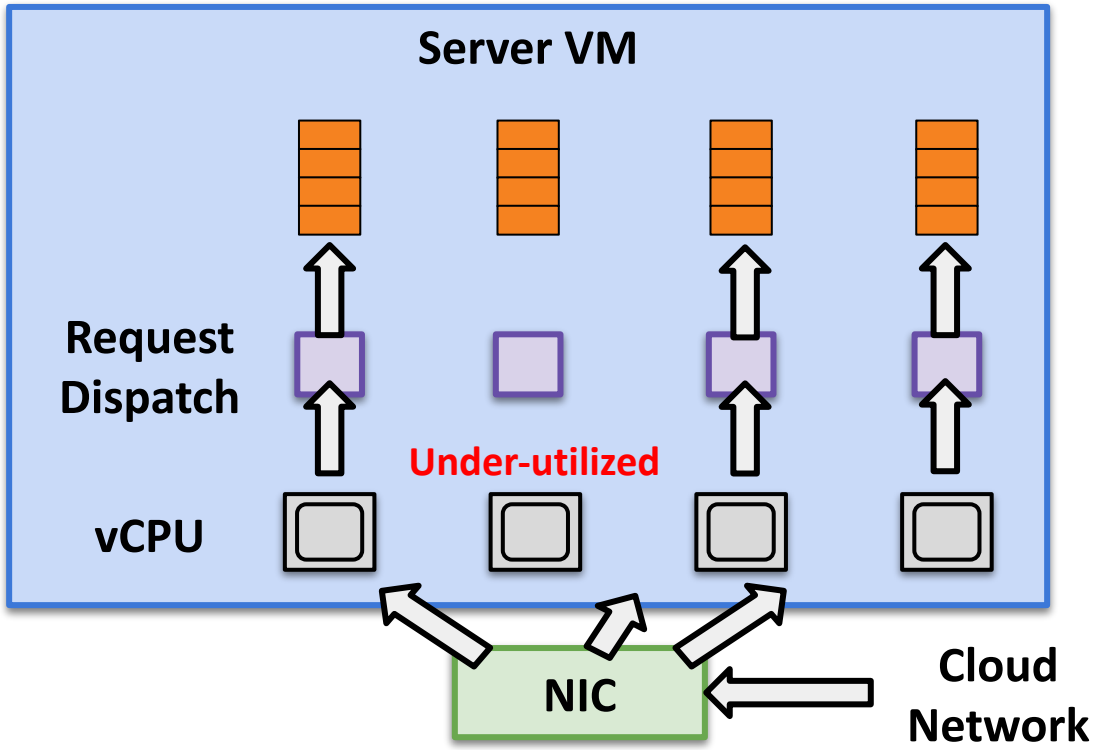
Seastar: Clients Could Route Requests Correctly



Clients could route to correct core/owner

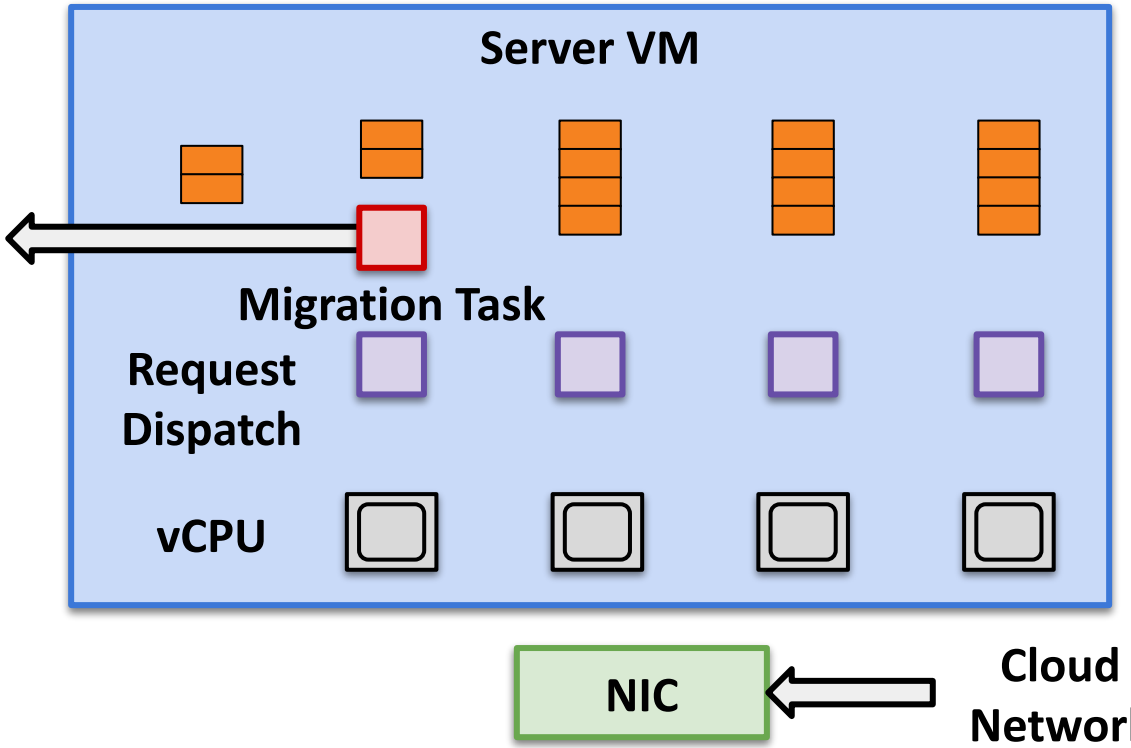
Quadratic connection state!
(client cores*server cores)
OR
Shuffle requests at client

Seastar: Bad For Skewed Workloads



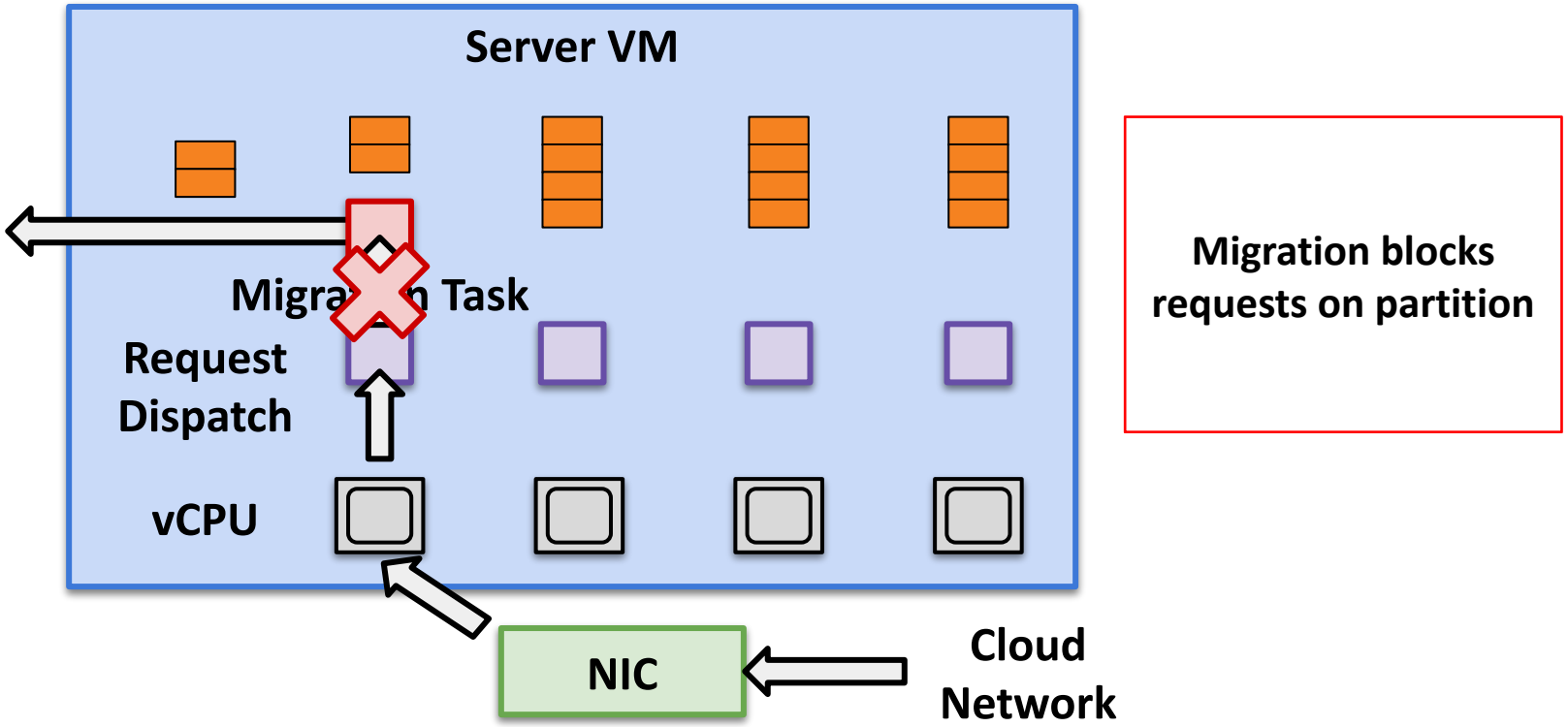
Bad for skewed workloads

Seastar: Migration Has To Be Single Threaded



Migration cannot be parallelized

Seastar: Head-of-line Blocking During Migration



Shadowfax: Design

Partitioned Request Dispatch, Shared Data

- ➔ vCPUs share lock-free index → record synchronization handled by cache coherence
- ➔ Migration can be parallelized, does not block requests on hash ranges

End-to-End Asynchronous Clients

- ➔ Issue pipelined asynchronous batches of requests to amortize network overhead
- ➔ Good for workloads with inter-request independence. Ex: click counts, heartbeats etc.

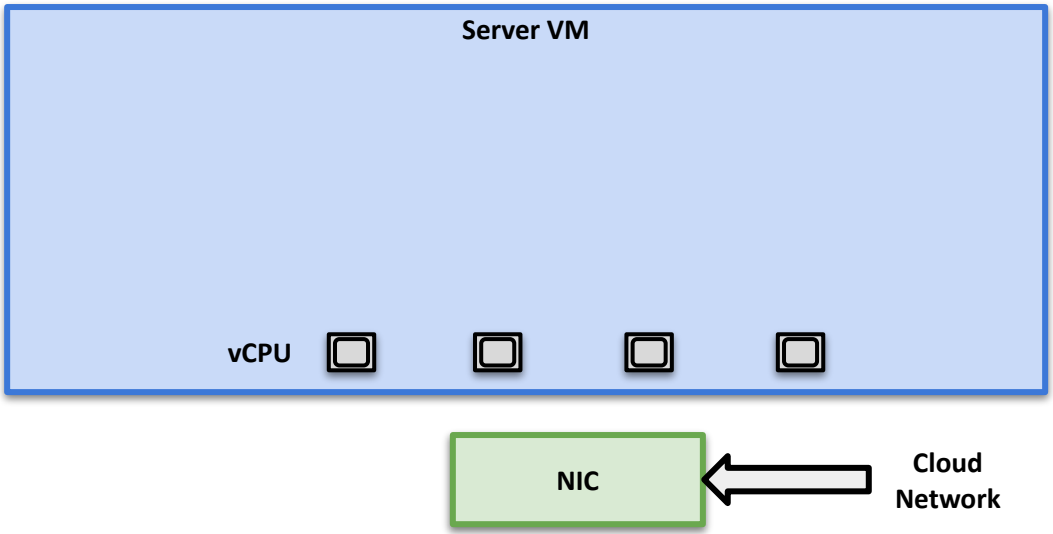
Asynchronous Global Cuts

- ➔ Per server view numbers represent hash ranges owned by server
- ➔ Cores avoid coordination during migration ownership changes

Shadowfax: Partitioned Request Dispatch Shared Data

Problem: **Avoid cross-core coordination at server**

Solution: **Offload all coordination to hardware cache-coherence protocol**



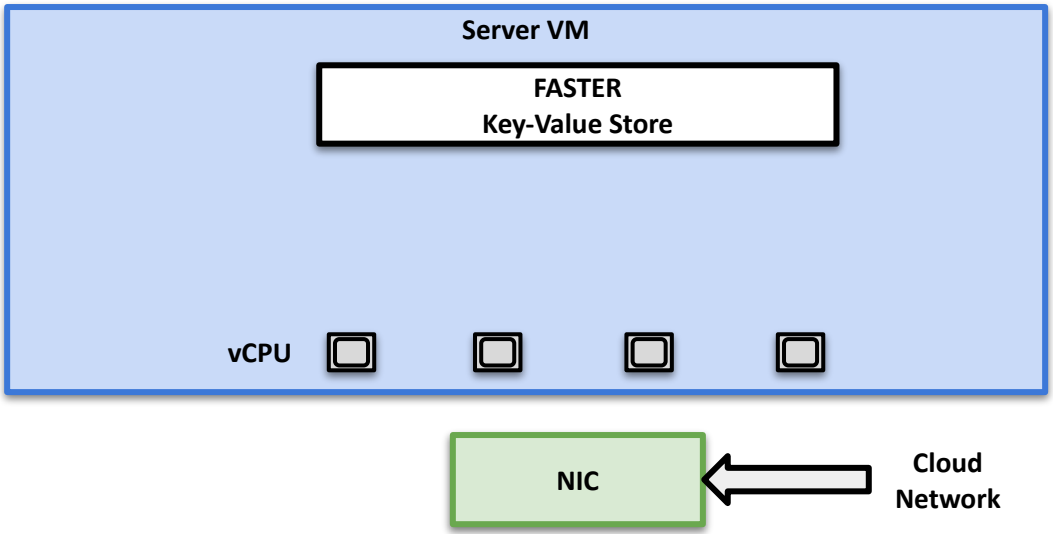
Shared memory is good for skew

Can serve hot records in parallel

Shadowfax: Partitioned Request Dispatch Shared Data

Problem: **Avoid cross-core coordination at server**

Solution: **Offload all coordination to hardware cache-coherence protocol**



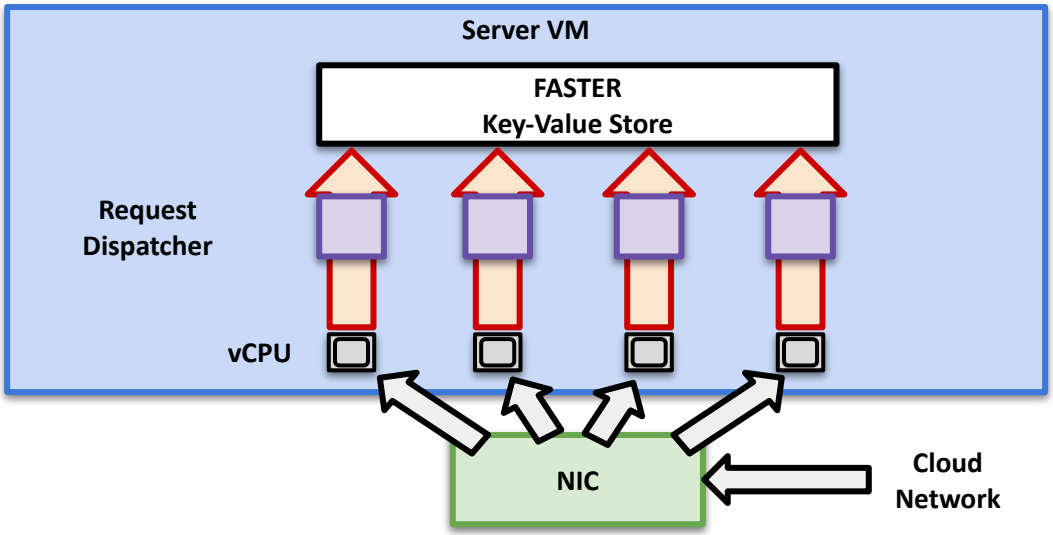
Lock-free index

Record synch happens at cache-coherence protocol (hw)

Shadowfax: Partitioned Request Dispatch Shared Data

Problem: **Multi-core request processing requires cross-core coordination**

Solution: **Offload all coordination to hardware cache-coherence protocol**



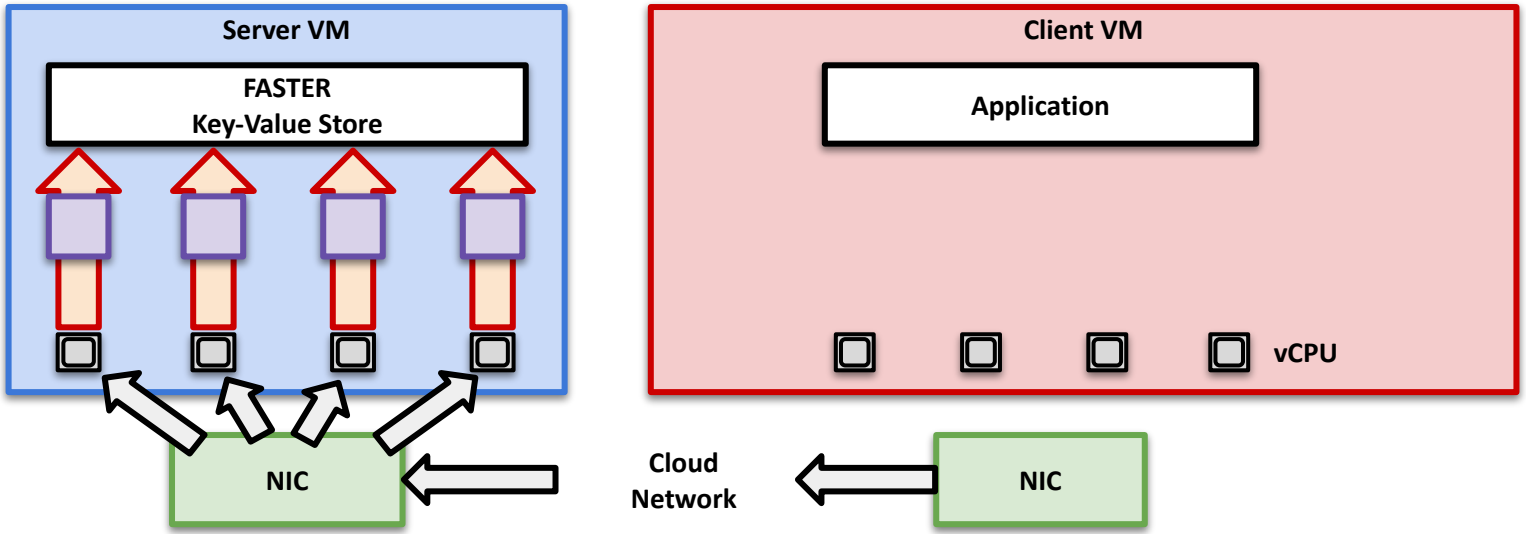
Partition request dispatch

Each dispatcher listens on separate TCP port

Shadowfax: Partitioned Request Dispatch Shared Data

Problem: **Keep servers saturated at index, not network or dispatch**

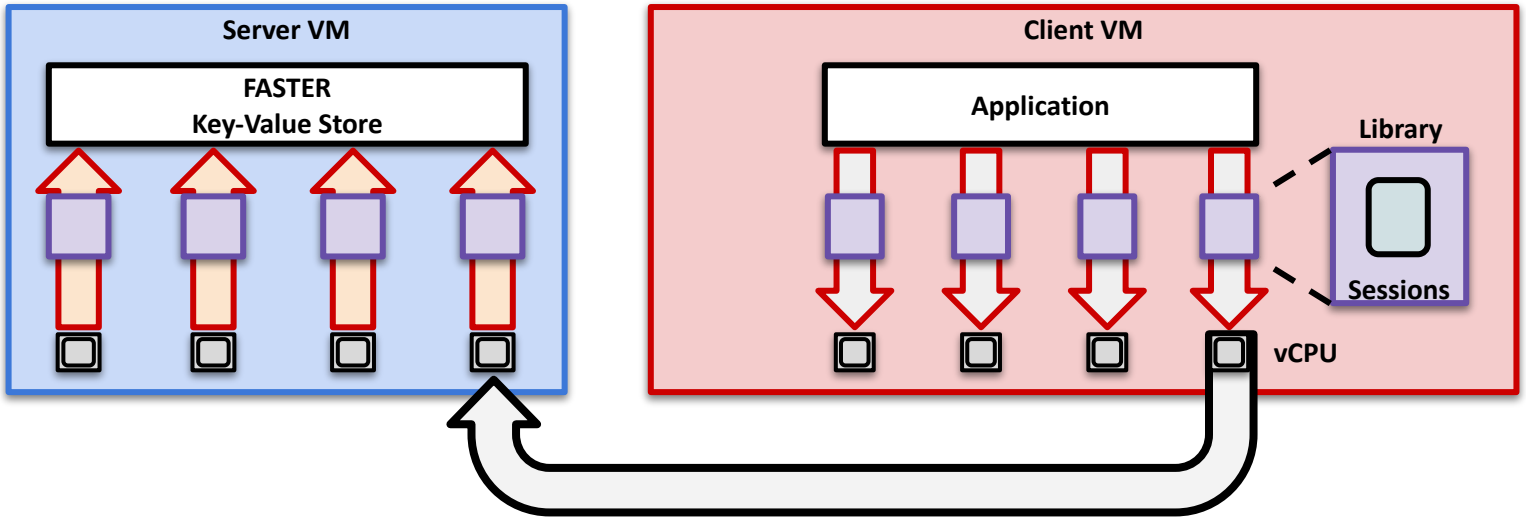
Solution: **Asynchronous client library, transparent network acceleration**



Shadowfax: Partitioned Request Dispatch Shared Data

Problem: Keep servers saturated at index, not network or dispatch

Solution: Asynchronous client library, transparent network acceleration

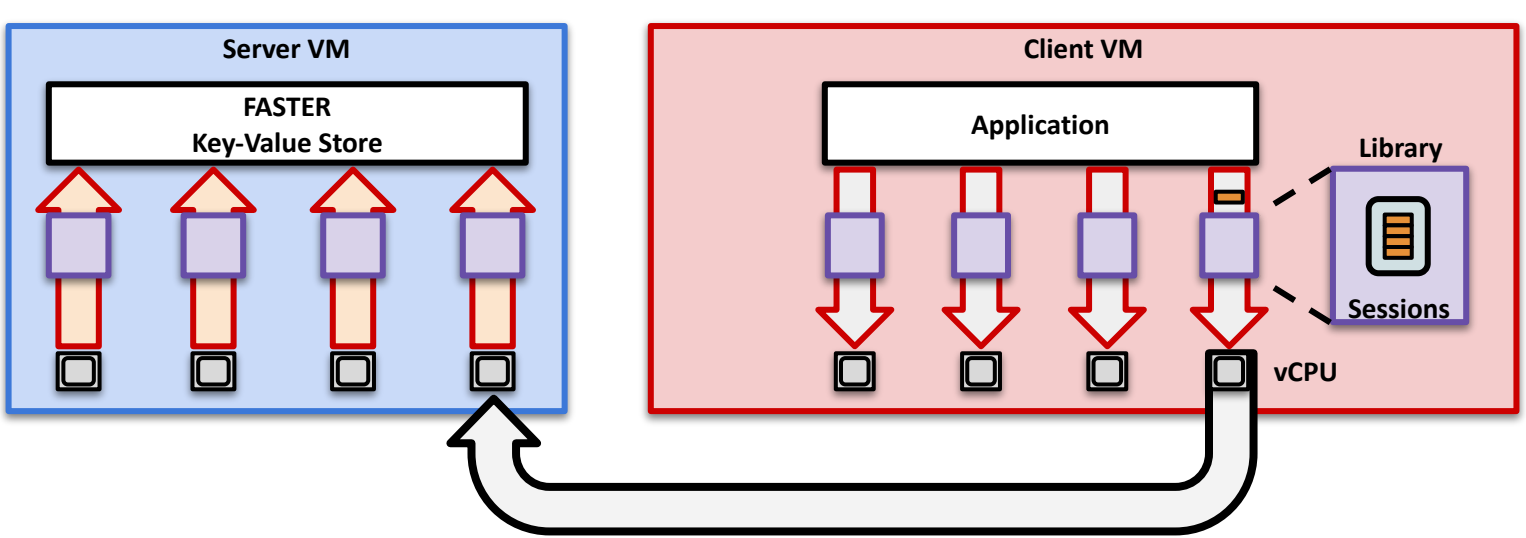


Client cores open *"sessions"* to server cores

Shadowfax: Partitioned Request Dispatch Shared Data

Problem: Keep servers saturated at index, not network or dispatch

Solution: Asynchronous client library, transparent network acceleration



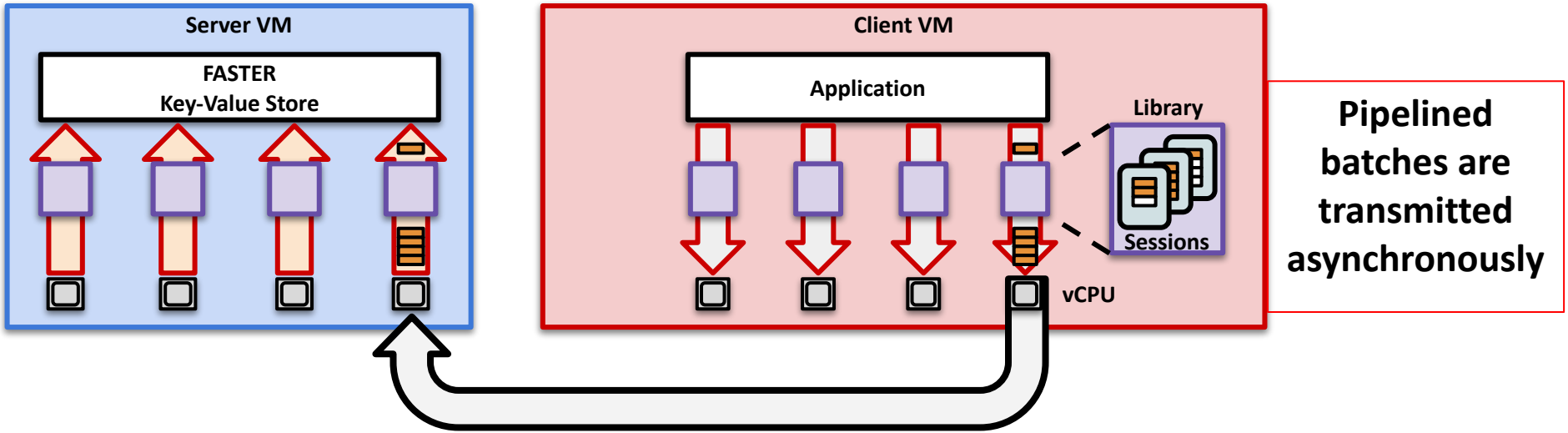
Requests are async

Enqueued inside session buffers

Shadowfax: Partitioned Request Dispatch Shared Data

Problem: Keep servers saturated at index, not network or dispatch

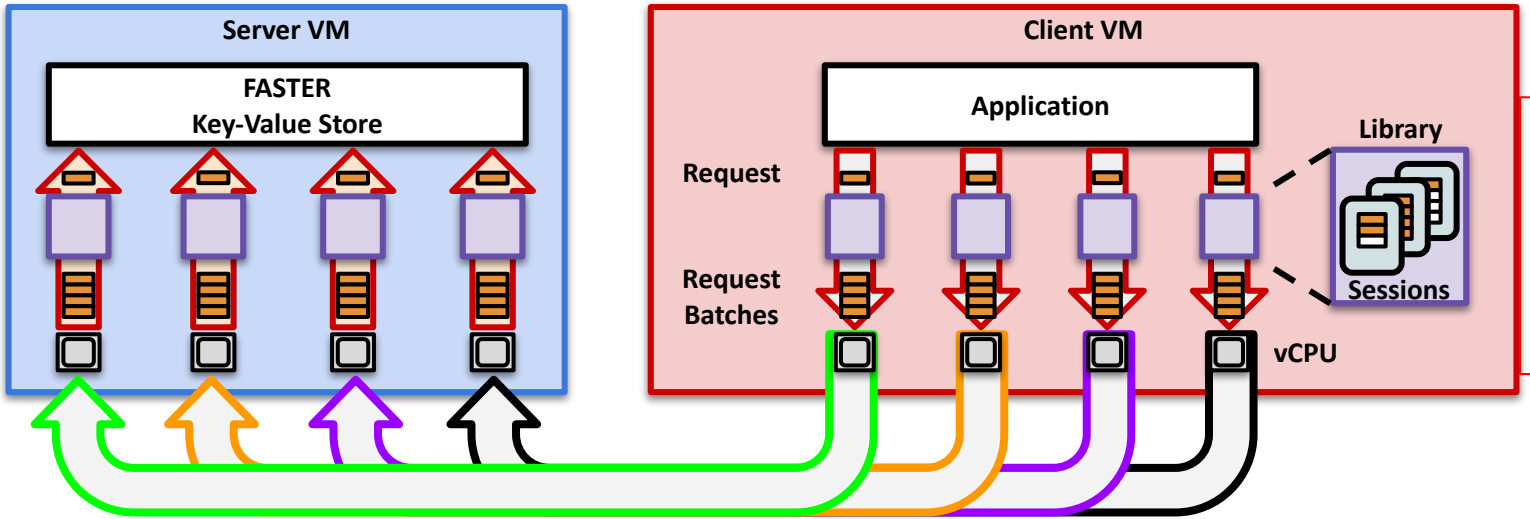
Solution: Asynchronous client library, transparent network acceleration



Shadowfax: Partitioned Request Dispatch Shared Data

Problem: Keep servers saturated at index, not network or dispatch

Solution: Asynchronous client library, transparent network acceleration

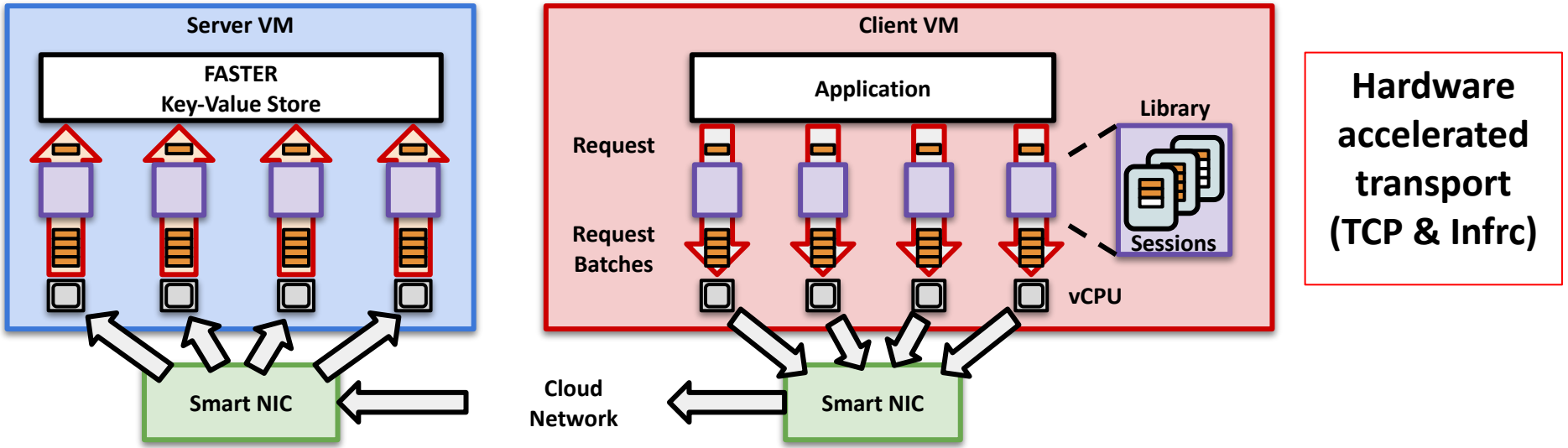


Pipelined batches are transmitted asynchronously

Shadowfax: Transparent Cloud Acceleration

Problem: Keep servers saturated at index, not network or dispatch

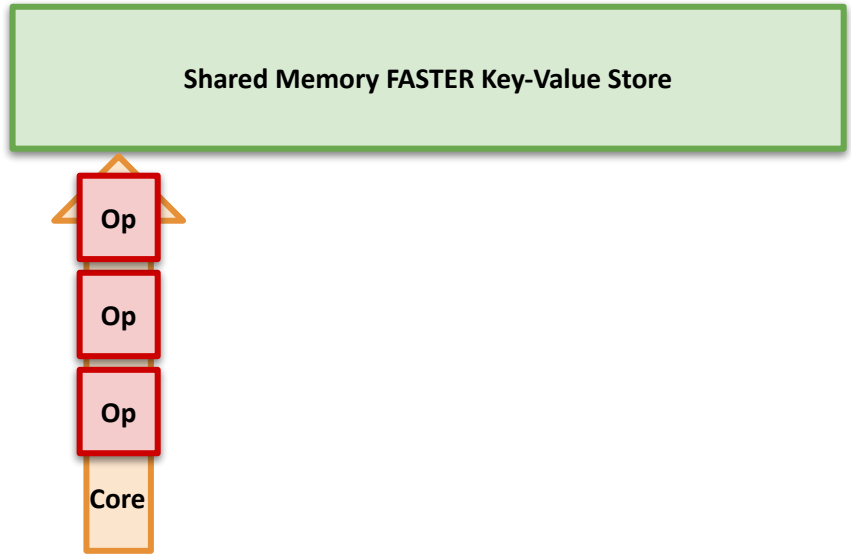
Solution: Asynchronous client library, transparent network acceleration



Shadowfax: Asynchronous Global Cuts

Problem: **Avoid cross-core coordination during migration**

Solution: **Server and client cores observe ownership change independently**



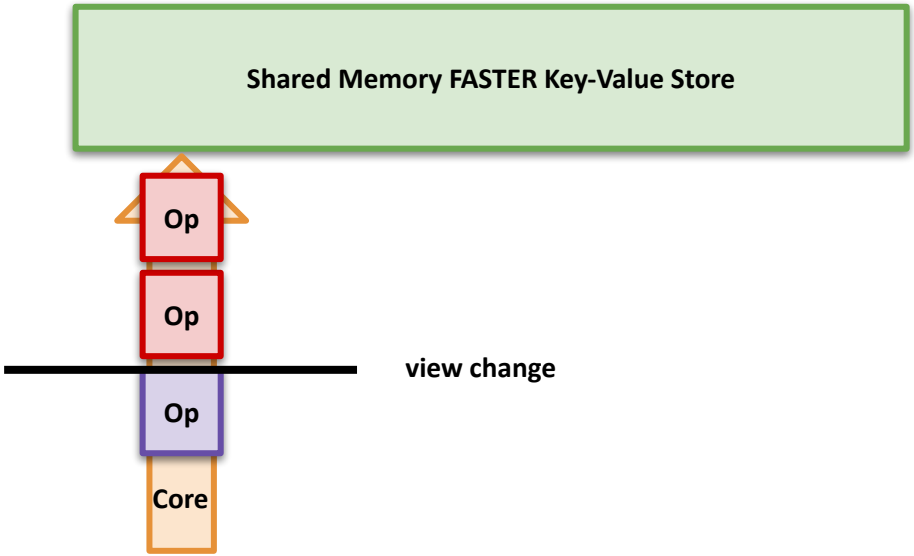
Migration changes server's ownership

"View" change

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently

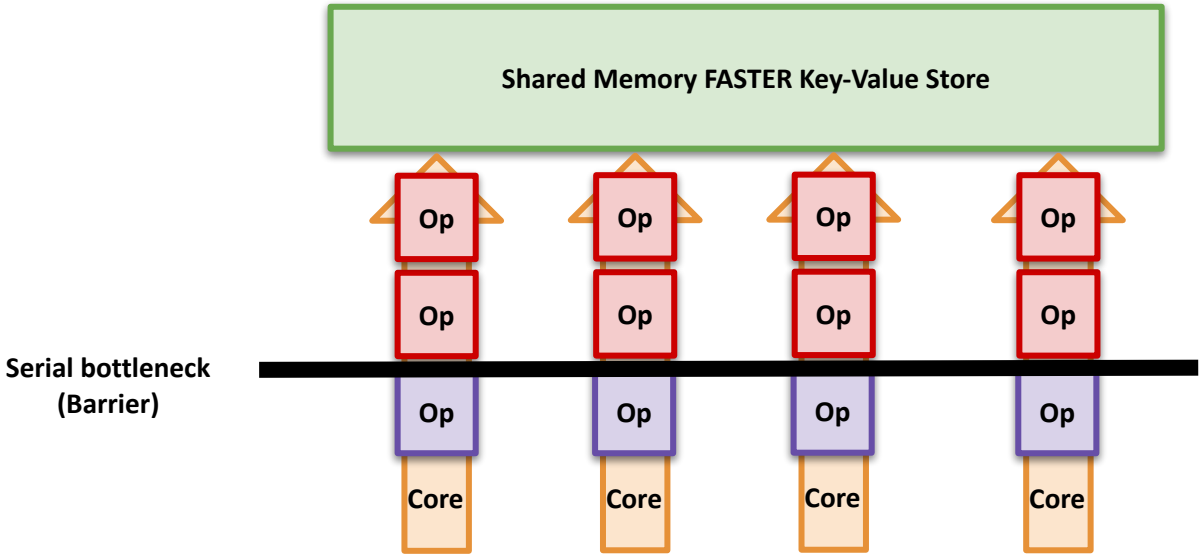


Operations after view change must be retransmitted to new owner

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently

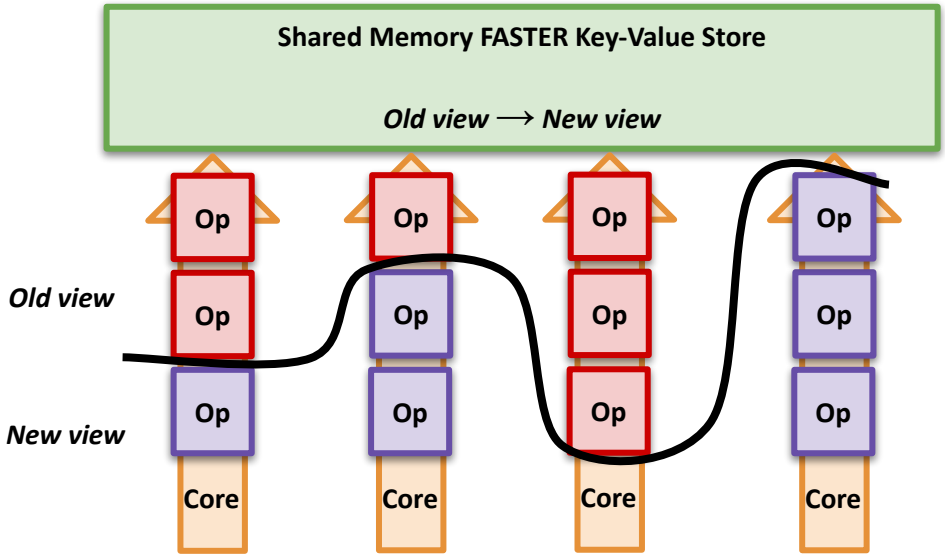


View change can become a serial bottleneck

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently



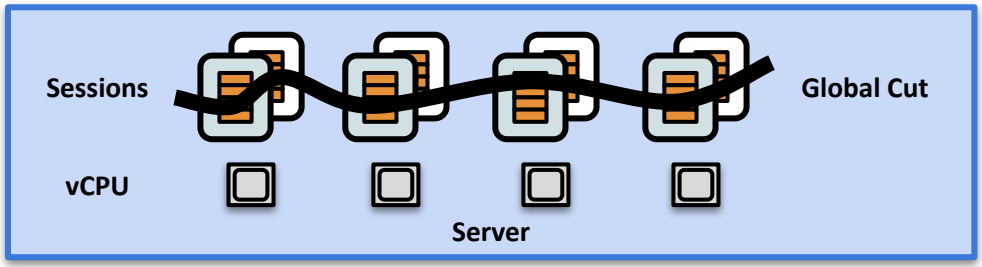
Each core observes change independently

"Async cut" avoids coordination within server

Shadowfax: Asynchronous Global Cuts

Problem: **Avoid cross-core coordination during migration**

Solution: **Server and client cores observe ownership change independently**

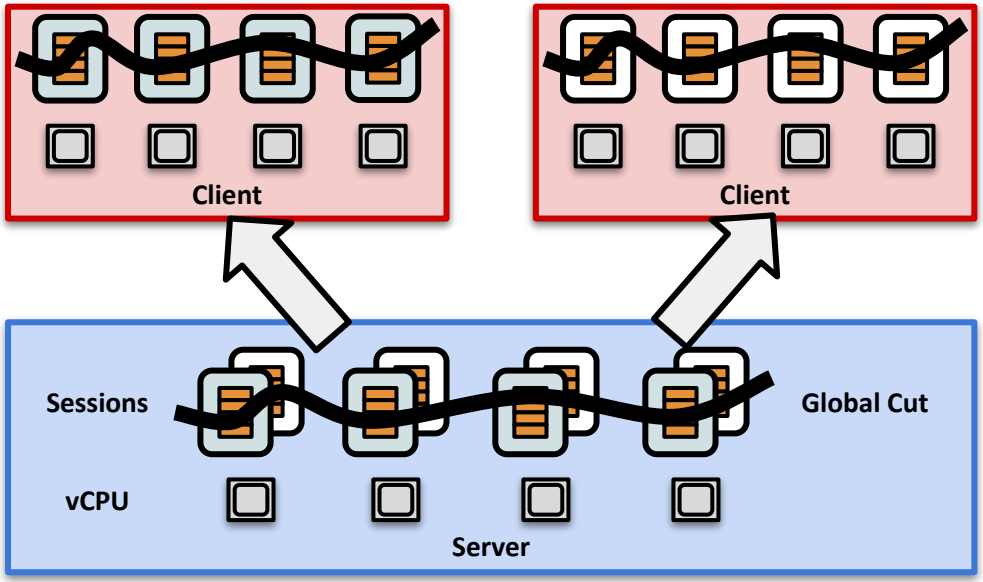


Async cut becomes *"global cut"* across sessions

Shadowfax: Asynchronous Global Cuts

Problem: Avoid cross-core coordination during migration

Solution: Server and client cores observe ownership change independently



Global cut pushed to clients

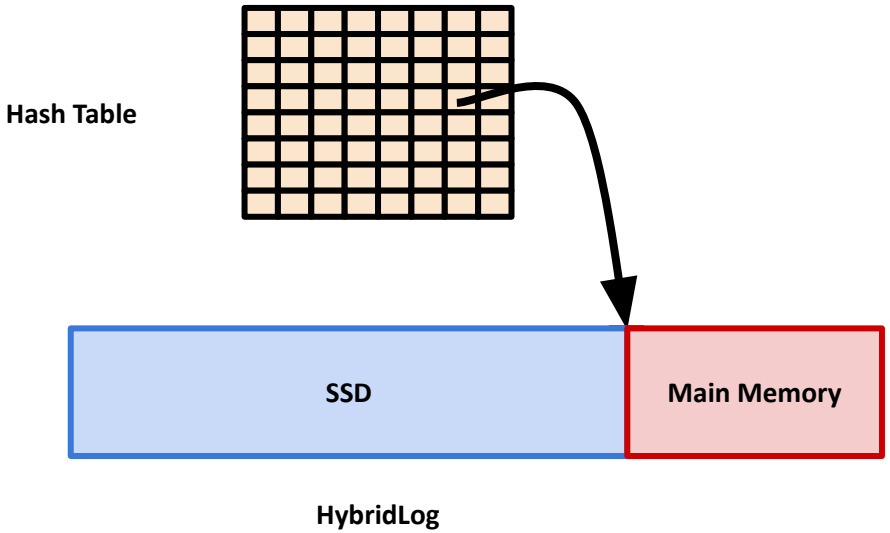
Avoids coordination at clients

Clients reissue operations

Shadowfax: Indirection Records

Problem: Migrating records on SSD can slow down reconfiguration

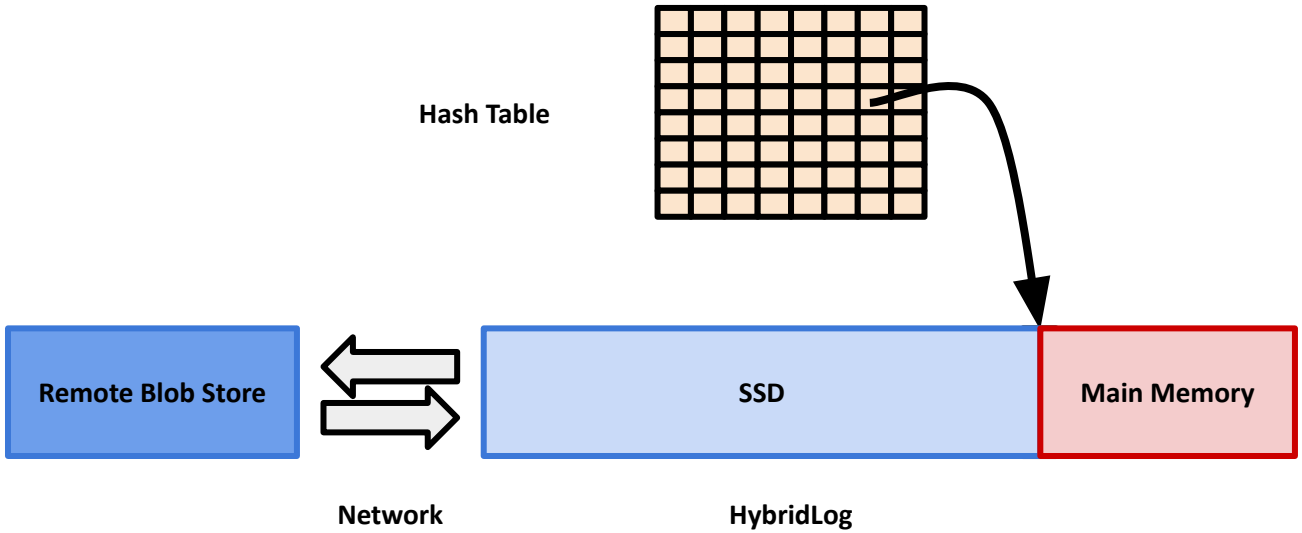
Solution: Use shared remote tier to restrict migration to main memory



Shadowfax: Indirection Records

Problem: Migrating records on SSD can slow down reconfiguration

Solution: Use shared remote tier to restrict migration to main memory

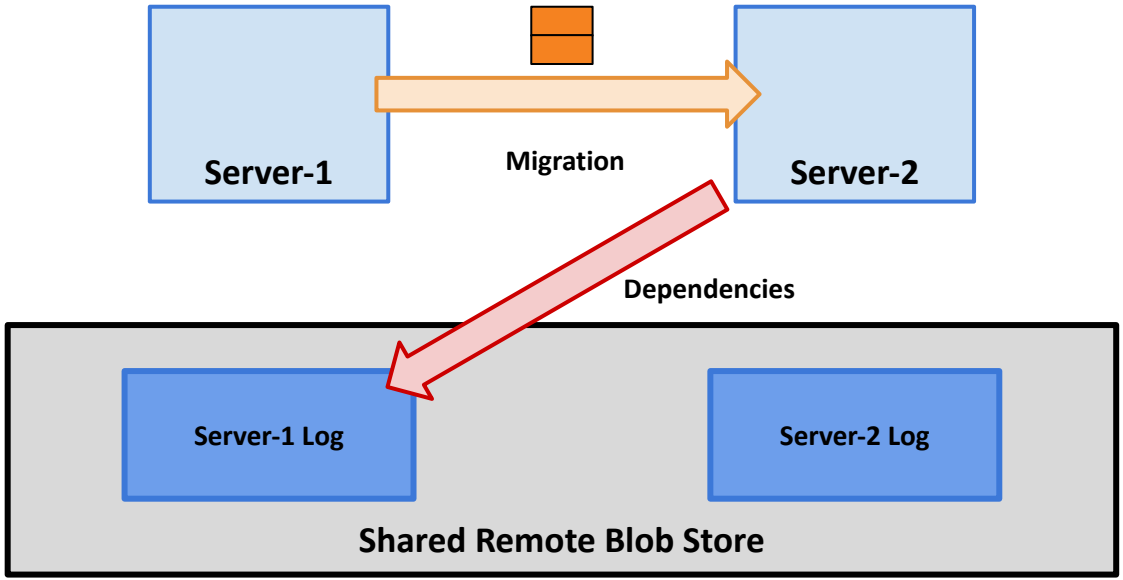


Records on SSD are also flushed to remote cloud blob storage

Shadowfax: Indirection Records

Problem: Migrating records on SSD can slow down reconfiguration

Solution: Use shared remote tier to restrict migration to main memory

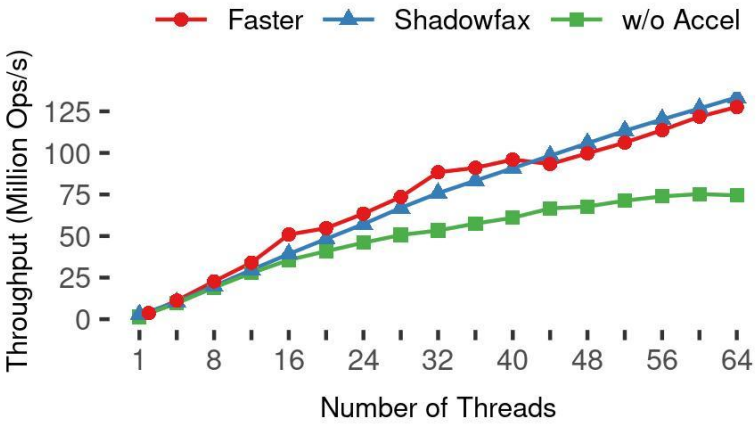


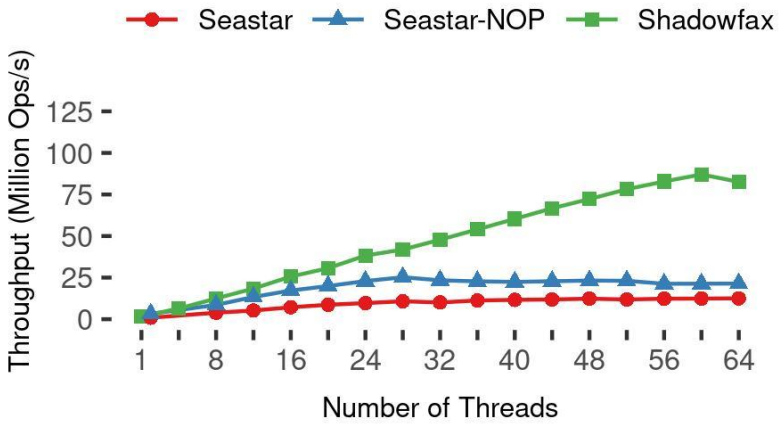
Migrate pointers to shared layer

Lazily cleaned up

Performance of Shadowfax

YCSB-F (Read-Modify-Writes, Benchmark Ingest of Events), 250 Million objects

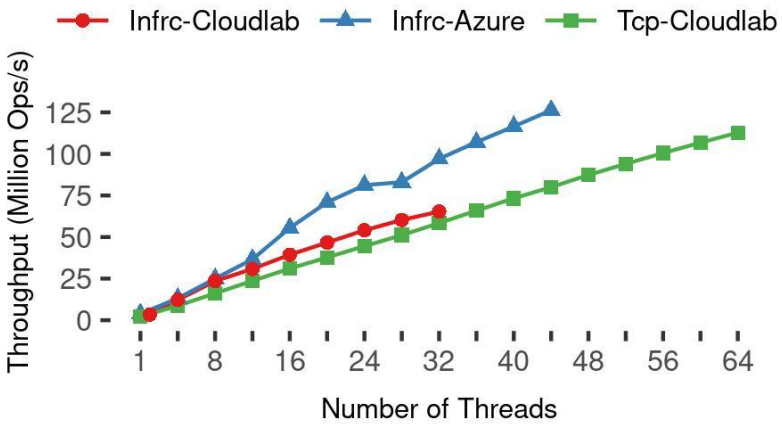




Saturates server at Index, 8.5x state-of-the-art

Performance of Shadowfax

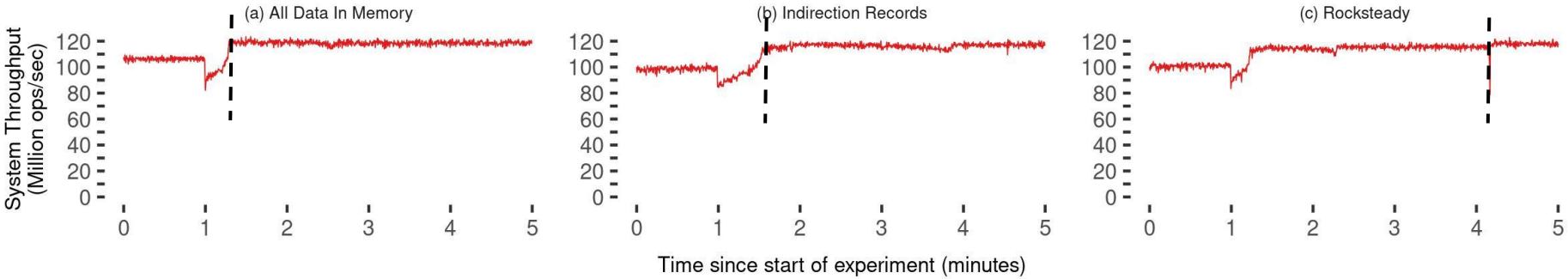
YCSB-F (Read-Modify-Writes, Benchmark Ingest of Events), 250 Million objects



Reproducible on different hardware platforms (Intel and AMD)

Performance of Shadowfax

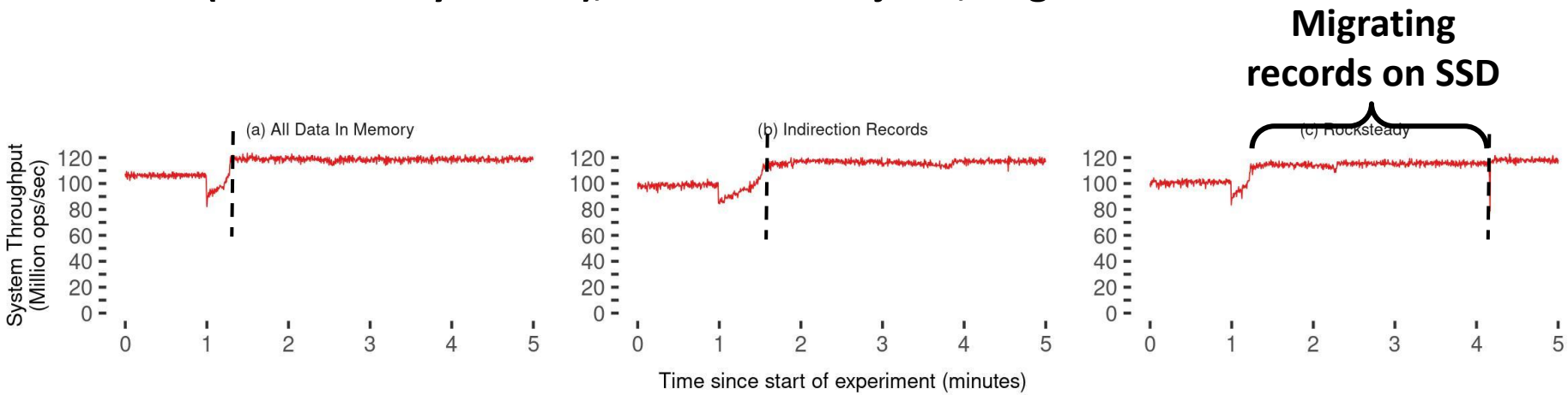
YCSB-F (Read-Modify-Writes), 250 Million objects, Migrate 10%



Throughput maintained > 80 Million events/second

Performance of Shadowfax

YCSB-F (Read-Modify-Writes), 250 Million objects, Migrate 10%



Indirection records complete migration 6x faster

Shadowfax Conclusion

Multi-core optimized single-node key-value stores are emerging

- ➔ **Very** high throughput ~**100 Million events/sec/server.** Ex: FASTER (SIGMOD'18)

Problem: Retain high throughput in the public cloud

- ➔ **Avoid dispatch and network bottlenecks** → Saturate servers within hash index
- ➔ Support reconfiguration/migration while preserving throughput

Solution: Partitioned Request Dispatch, Asynchronous Global Cuts

- ➔ Cores share FASTER index, dispatch layer is partitioned → **retain high throughput**
- ➔ Cores avoid coordination during migration → **maintain high throughput**

Shadowfax: High Throughput, Elastic, Larger-than-Memory key-value store

- ➔ **100 Million events/sec/VM on Azure**, **80 Million events/sec/VM** during migration
- ➔ Spans DRAM, SSD and Cloud blob storage

Extensibility and Multi-tenancy

Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. **Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage.** In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, **OSDI'18**

Ankit Bhardwaj, **Chinmay Kulkarni**, and Ryan Stutsman. **Adaptive Placement for In-memory Storage Functions.** In *Proceedings of the 2020 USENIX Annual Technical Conference,* **ATC'20**

Splinter Introduction

Kernel-bypass key-value stores offer **< 10µs** latency**, > Mops/s** throughput

- ➔ Fast because they support just simple gets and puts?

Problem: Leverage performance ⟶ **share** between tenants

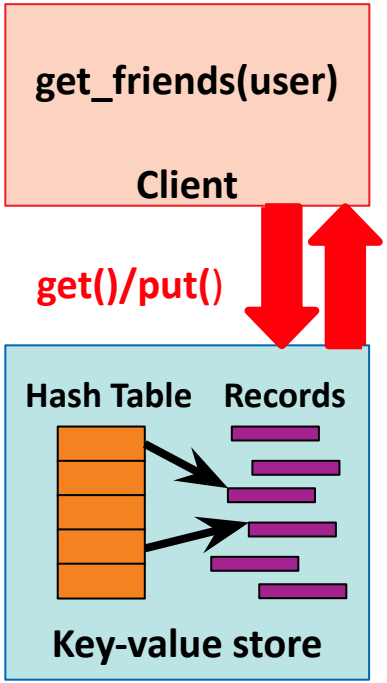
Problem: Apps require rich data models. Ex: Facebook's TAO

- ➔ Implement using gets & puts? ⟶ **Data movement, client stalls**
- ➔ Push code to key-value store? ⟶ **Isolation costs limit density**

Splinter: **Multi-tenant** key-value store that code can be pushed to

- ➔ Tenants push type- & memory-safe code written in **Rust** at runtime
- ➔ Aggregate workload improves by **2x**, Facebook's TAO by **400 Kops/s**

Richer Data Models Come At A Price



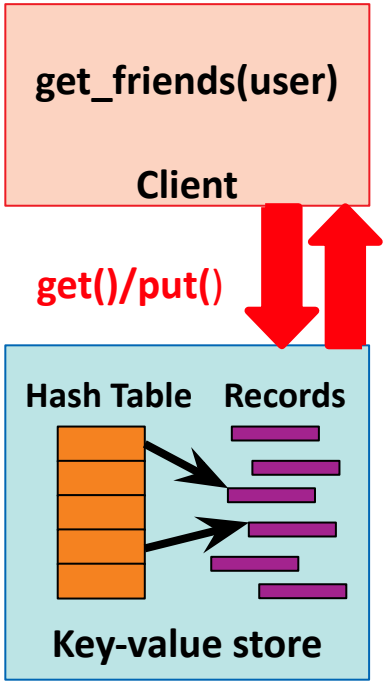
Apps require rich data models in addition to performance

➔ Ex: Social graphs, Decision trees etc

Key-value stores trade-off data model for performance

➔ Simple get()'s & put()'s over key-value pairs

Richer Data Models Come At A Price



Apps require rich data models in addition to performance

➔ Ex: Social graphs, Decision trees etc

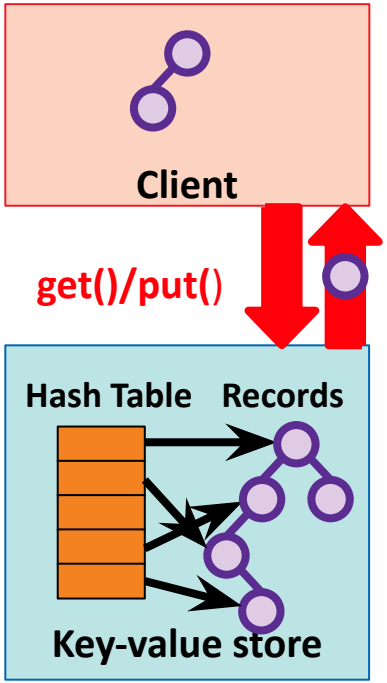
Key-value stores trade-off data model for performance

➔ Simple get()'s & put()'s over key-value pairs

Thinner data model → Better performance

But do applications benefit?

Extra Round-Trips (RTTs) Hurt Latency & Utilization



Example: Traverse tree with N nodes using gets

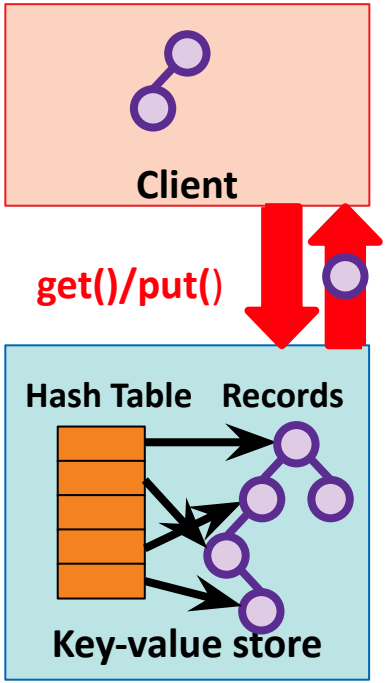
→ One get() at each level of the tree ⟶ **O(log N) RTTs**

→ Control flow depends on data ⟶ **Client stalls during get()**

Network RTTs, dispatch are the main bottleneck ~10µs

→ 1.5µs inside the server

Extra Round-Trips (RTTs) Hurt Latency & Utilization



Example: Traverse tree with N nodes using gets

→ One get() at each level of the tree ⟶ **O(log N) RTTs**

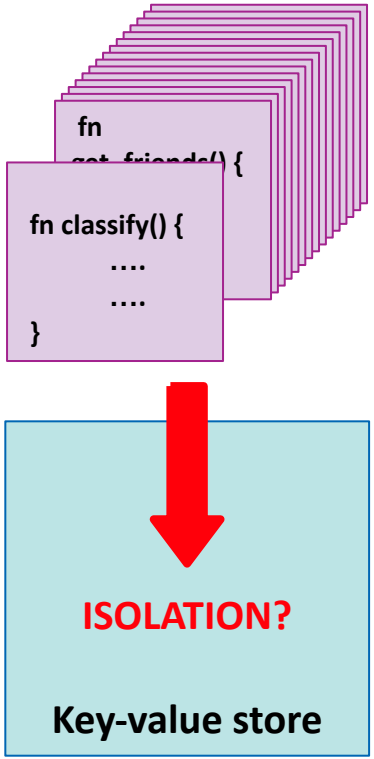
→ Control flow depends on data ⟶ **Client stalls during get()**

Network RTTs, dispatch are the main bottleneck ~10µs

→ 1.5µs inside the server

So push code to storage?

Why Not Push Compute To Storage?



RPC Processing Time ~1.5µs

Only native code will do

Context Switches ~1.5µs

Multi-tenancy → Need hardware isolation

What Do We Want From The Storage Layer?

Granularity of compute is steadily decreasing

Virtual machines → Containers → Lambdas

Extremely high tenant density

➔ Fine-grained resource allocation; 100s of CPU cycles, Kilobytes of memory

Allow tenants to extend data model at runtime

➔ Low overhead isolation between tenants & storage layer

Splinter: A Multi-Tenant Key-Value Store

Tenants can install and invoke extensions at runtime

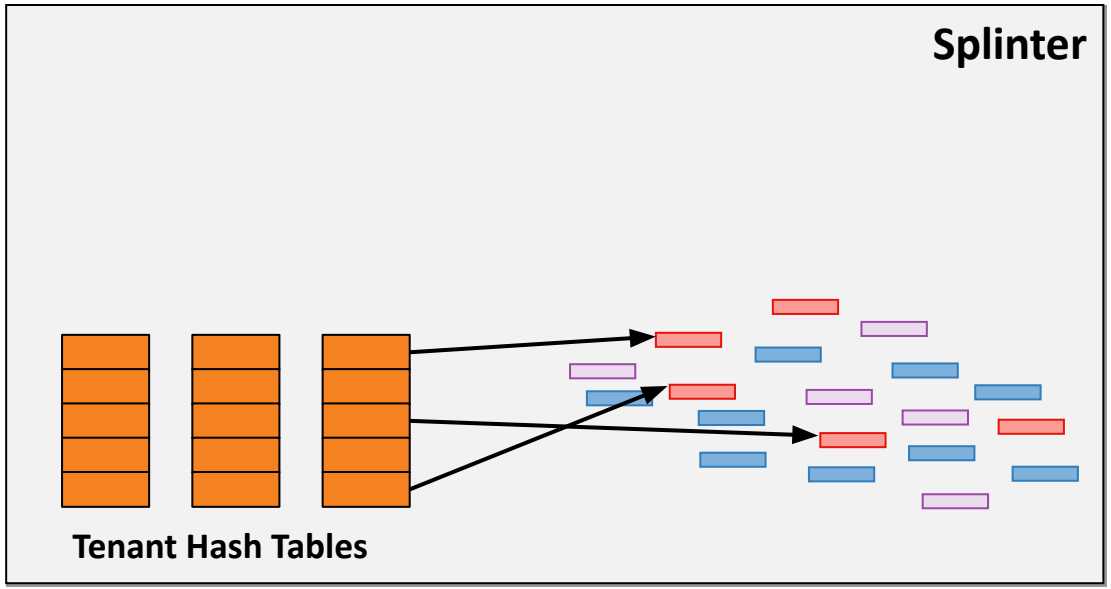
- → Extensions written in **Rust**
- → Rely on type and memory safety for isolation, avoids context switch

Implemented in ~9000 lines of Rust

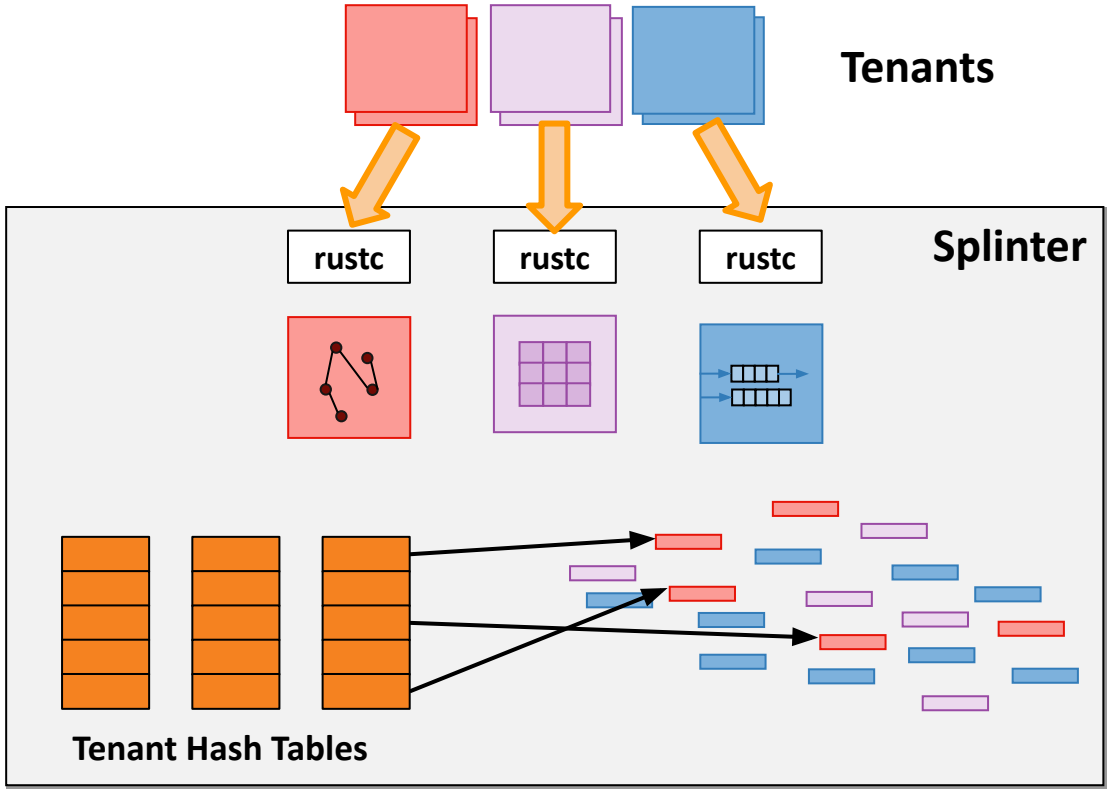
- → Supports two RPCs → **install(ext_name) & invoke(ext_name)**
- → Also supports regular get() & put() RPCs → **“Native” operations**

1000 Foot View Of Splinter

Tenants



1000 Foot View Of Splinter

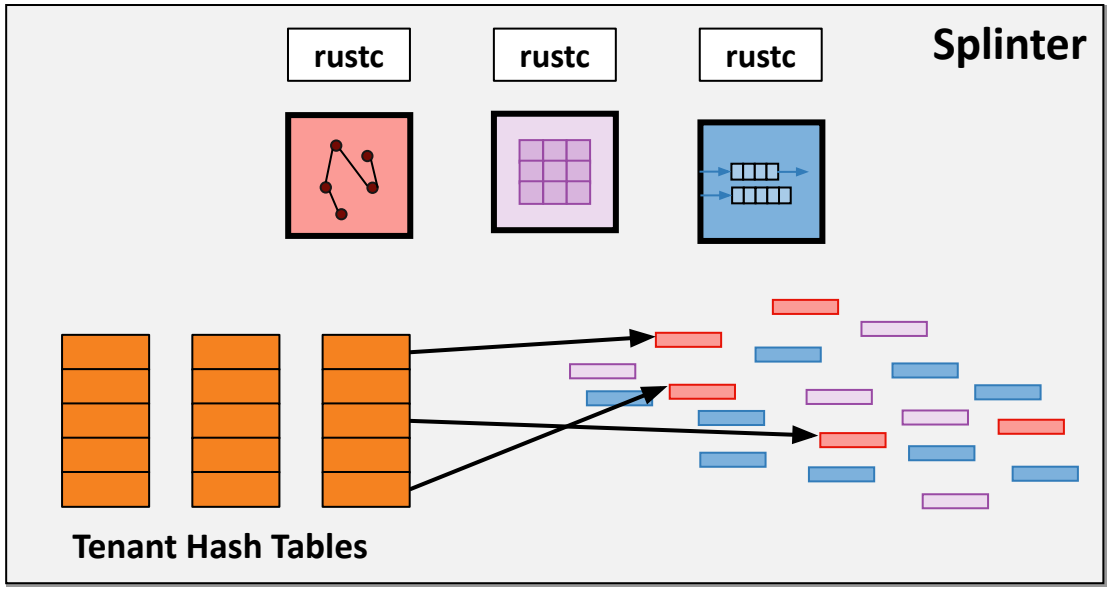


Tenants push extensions written in Rust

Splinter loads extensions Into address-space

1000 Foot View Of Splinter

Tenants

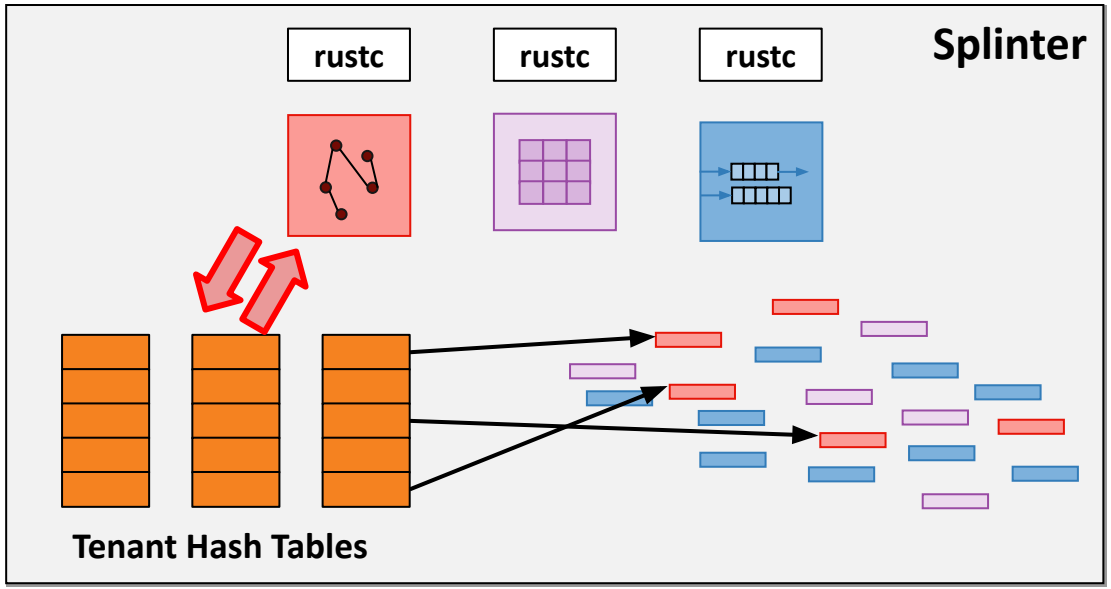


Rust provides memory-safety

Extensions do not share state

1000 Foot View Of Splinter

Tenants



Extensions receive references to records

Each tenant sees a custom key-value store

Splinter: Design

Low cost isolation

→ No forced data copies across trust boundary

Tenant Locality And Work Stealing

→ Avoid cross-core coordination while avoiding hotspots

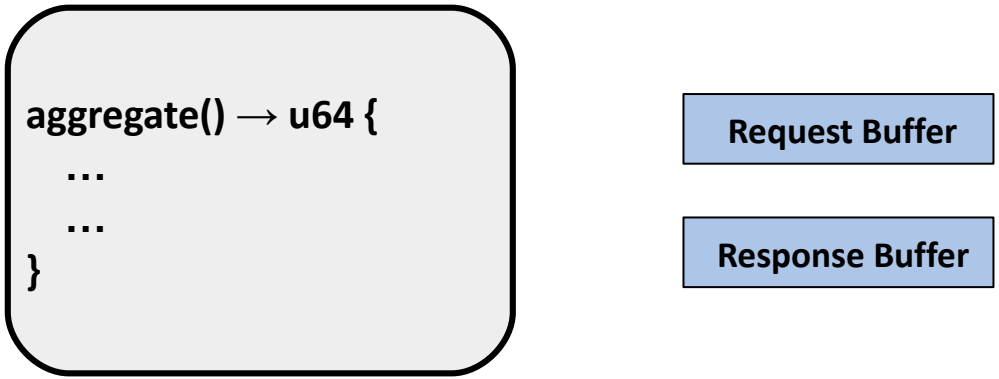
Lightweight Cooperative Scheduling

→ Prevent long running extensions from starving short running ones

Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

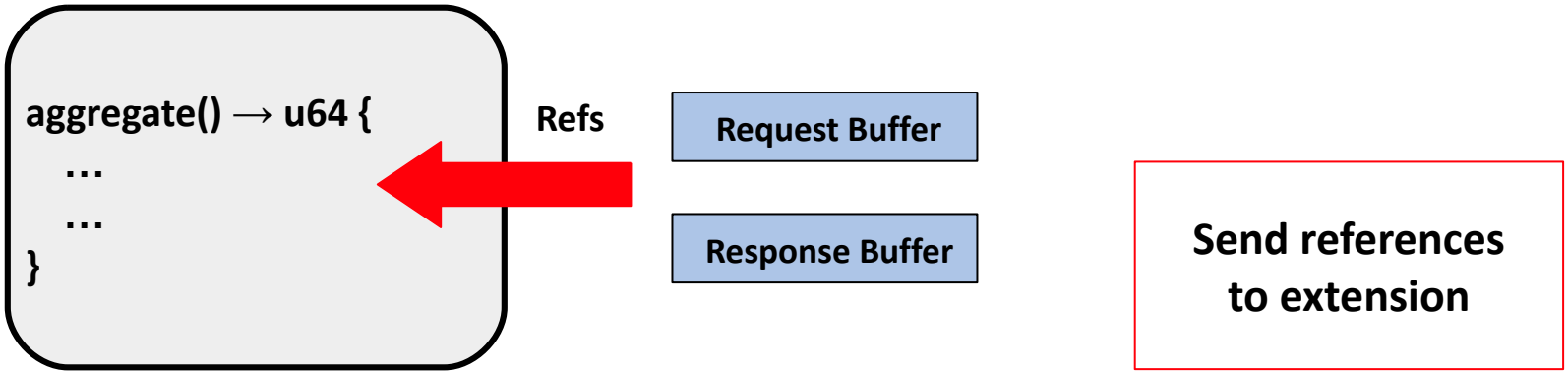
Solution: **Ensure buffers outlast reference lifetime**



Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

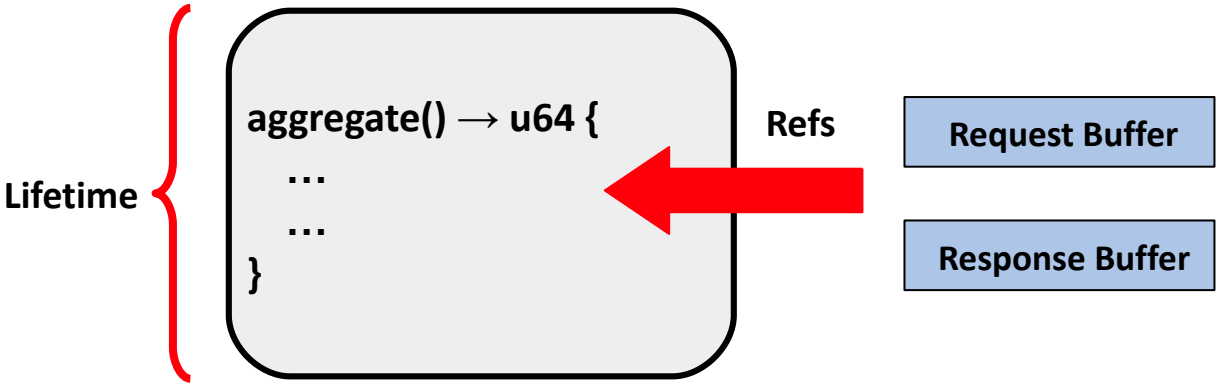
Solution: **Ensure buffers outlast reference lifetime**



Splinter: Low Cost Isolation

Problem: **No forced data copies across trust boundary**

Solution: **Ensure buffers outlast reference lifetime**

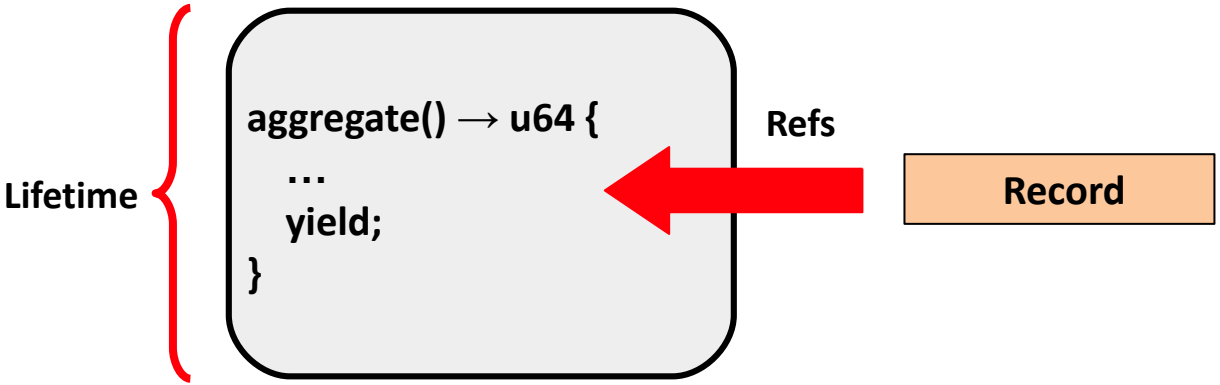


Statically ensure RPC buffers outlast lifetime

Splinter: Low Cost Isolation

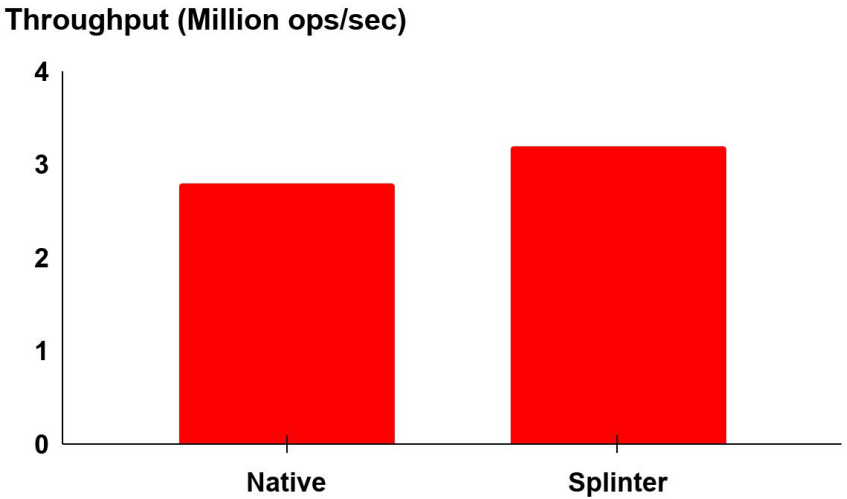
Problem: **No forced data copies across trust boundary**

Solution: **Ensure buffers outlast reference lifetime**



Statically ensure record stays stable across yields

Pushing Facebook's TAO To Splinter



Hybrid → get() for point ops, extension for dependencies

Best of both worlds!

Splinter Conclusion

Kernel-bypass key-value stores offer **< 10µs** latency**, > Mops/s** throughput

- ➔ Fast because they support just simple gets and puts?

Problem: Leverage performance ⟶ **share** between tenants

Problem: Apps require rich data models. Ex: Facebook's TAO

- ➔ Implement using gets & puts? ⟶ **Data movement, client stalls**
- ➔ Push code to key-value store? ⟶ **Isolation costs limit density**

Splinter: **Multi-tenant** key-value store that code can be pushed to

- ➔ Tenants push type- & memory-safe code written in **Rust** at runtime
- ➔ Aggregate workload improves by **2x**, Facebook's TAO by **400 Kops/s**

Thesis Contributions

Fast Data Migration with Low Latency Impact [Rocksteady, SOSP'17]

Low-cost Coordination for Request Dispatch and Migration [Shadowfax, VLDB'21, Revision]

Extensibility and Multi-tenancy without Hardware Isolation [Splinter, OSDI'18]

Low-latency stores adopt simple, stripped down designs that optimize for normal case performance, and in the process, trade off features that would make them more practical and cost effective at cloud scale. This thesis shows that this trade off is unnecessary. Carefully leveraging and extending new and existing abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their primary performance benefits at cloud scale.

Thesis Contributions

Fast Data Migration with Low Latency Impact [Rocksteady, SOSP'17]

Low-cost Coordination for Request Dispatch and M[illegible]tion [Shadowfax,

Extensibility and Multi-tenancy without Hardware Isolatio[illegible] [Splinter, OSDI'18]

Create headroom

On-demand migration of hot keys

Low-latency stores adopt simple, stripped down designs that o[illegible]timize for normal case performance, and in the process, trade off features that would ma[illegible] them more practical and cost effective at cloud scale. This thesis shows that this trade[illegible]ff is unnecessary. Carefully leveraging and extending new and existing abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their primary performance benefits at cloud scale.

Thesis Contributions

Fast Data Migration with Low Latency Impact [Rocksteady, SOSP'17]

Low-cost Coordination for Request Dispatch and Migration [Shadowfax,

Extensibility and Multi-tenanc thout Hardware Isolation [Splinter, OSDI'18]

Partitioned Dispatch Shared Data

Asynchronous Global Cuts

Low-latency stores a pt simple, stripped down designs that optimize for normal case performance, and the process, trade off features that would make them more practical and cost effecti at cloud scale. This thesis shows that this trade off is unnecessary. Carefully lev ging and extending new and existing abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their primary performance benefits at cloud scale.

Thesis Contributions

Fast Data Migration with Low Latency Impact [Rocksteady, SOSP'17]

Low-cost Coordination for Request Dispatch and Migration [Shadowfax,

Extensibility and Multi-tenancy without Hardware Isolation [Splinter, OSDI'18]

Low-cost Isolation

Co-op scheduling
Tenant Locality

Low-latency stores ad simple, str down designs that optimize for normal case performance, and in the ocess, trade off featu would make them more practical and cost effective at clou scale. This thesis shows that de off is unnecessary. Carefully leveraging and ex nding new and existing abstractions fo scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their primary performance benefits at cloud scale.

Thesis Contributions

Fast Data Migration with Low Latency Impact [Rocksteady, SOSP'17]

Low-cost Coordination for Request Dispatch and Migration [Shadowfax, VLDB'21, Revision]

Extensibility and Multi-tenancy without Hardware Isolation [Splinter, OSDI'18]

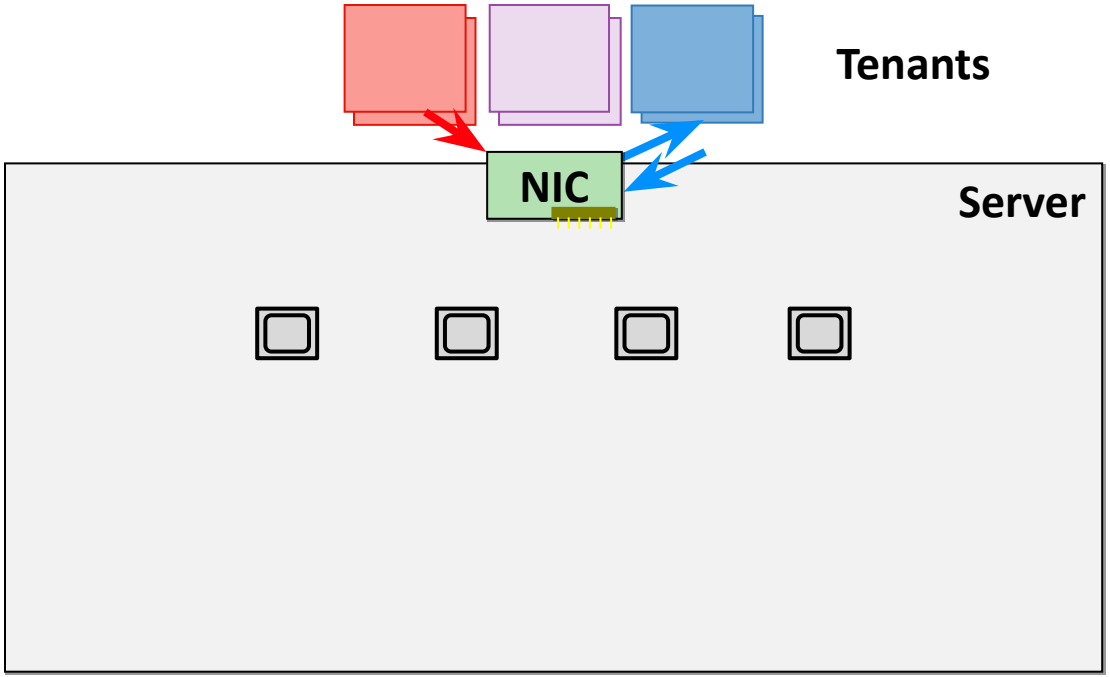
Low-latency stores adopt simple, stripped down designs that optimize for normal case performance, and in the process, trade off features that would make them more practical and cost effective at cloud scale. This thesis shows that this trade off is unnecessary. Carefully leveraging and extending new and existing abstractions for scheduling, data sharing, lock-freedom, and isolation will yield feature-rich systems that retain their primary performance benefits at cloud scale.

Backup Slides

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

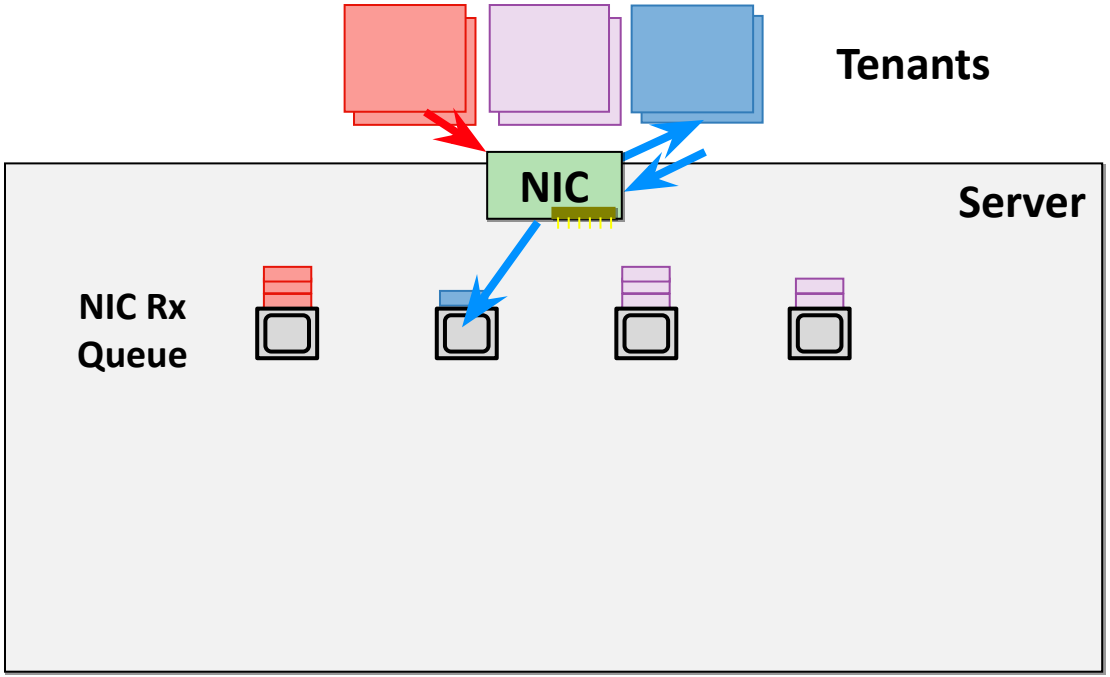
Solution: **NIC routes tenants to cores, cores steal work**



Splinter: Tenant Locality And Work Stealing

Problem: Quickly dispatch requests to cores, avoid hotspots

Solution: NIC routes tenants to cores, cores steal work

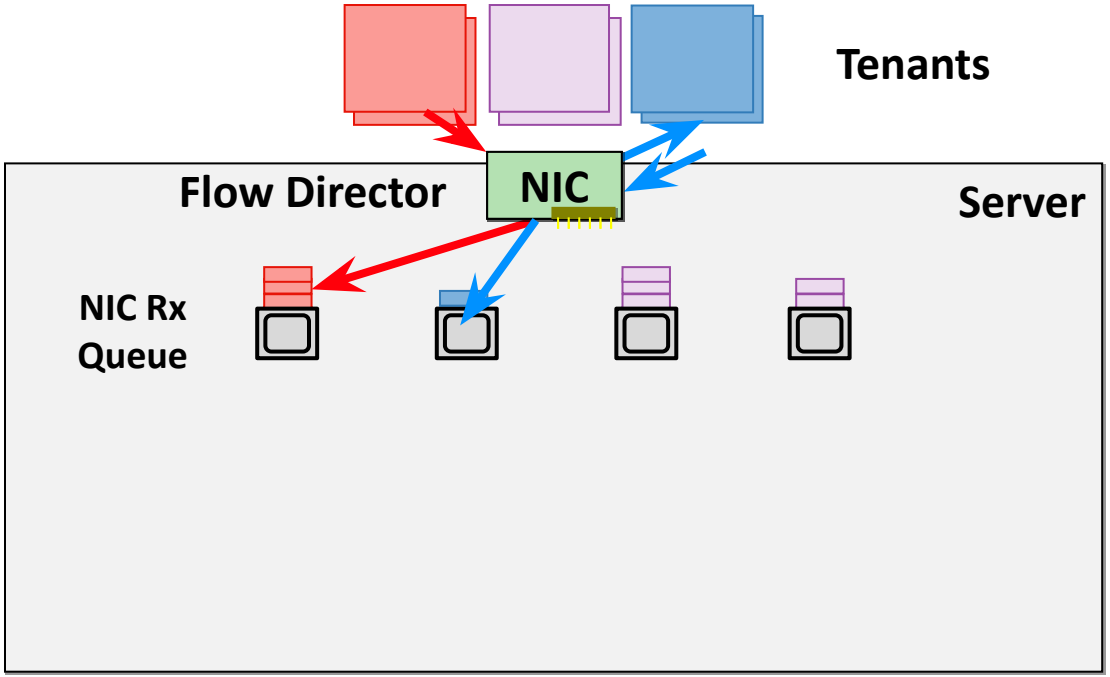


One Rx queue per core

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**

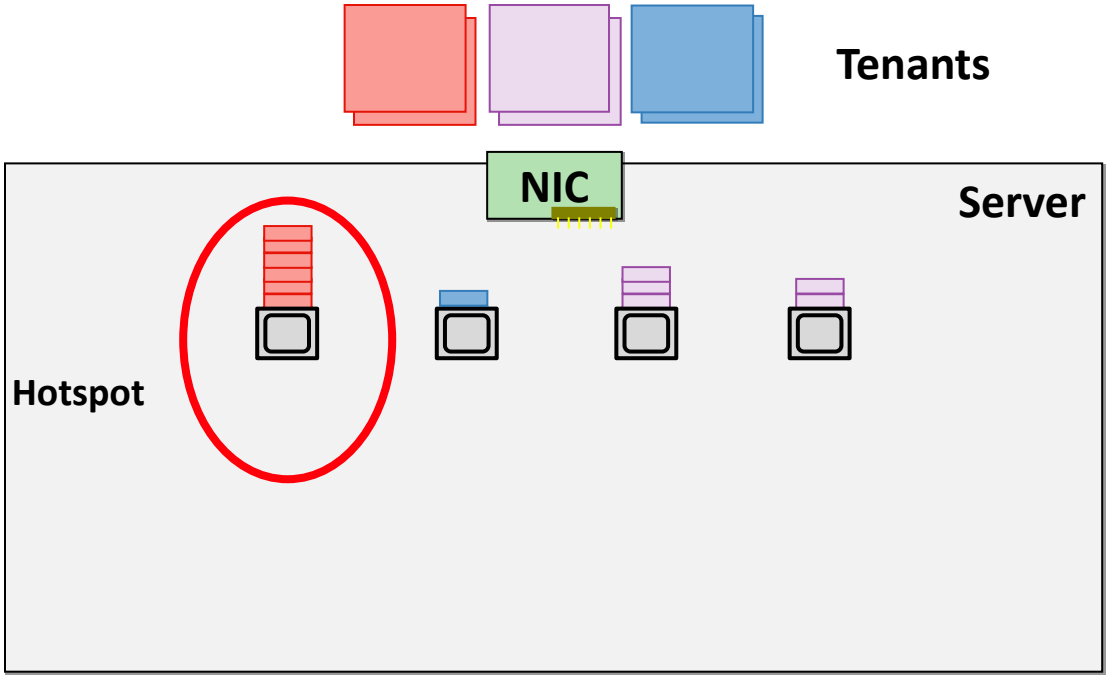


Maintain “Locality”
route tenant to queue

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**

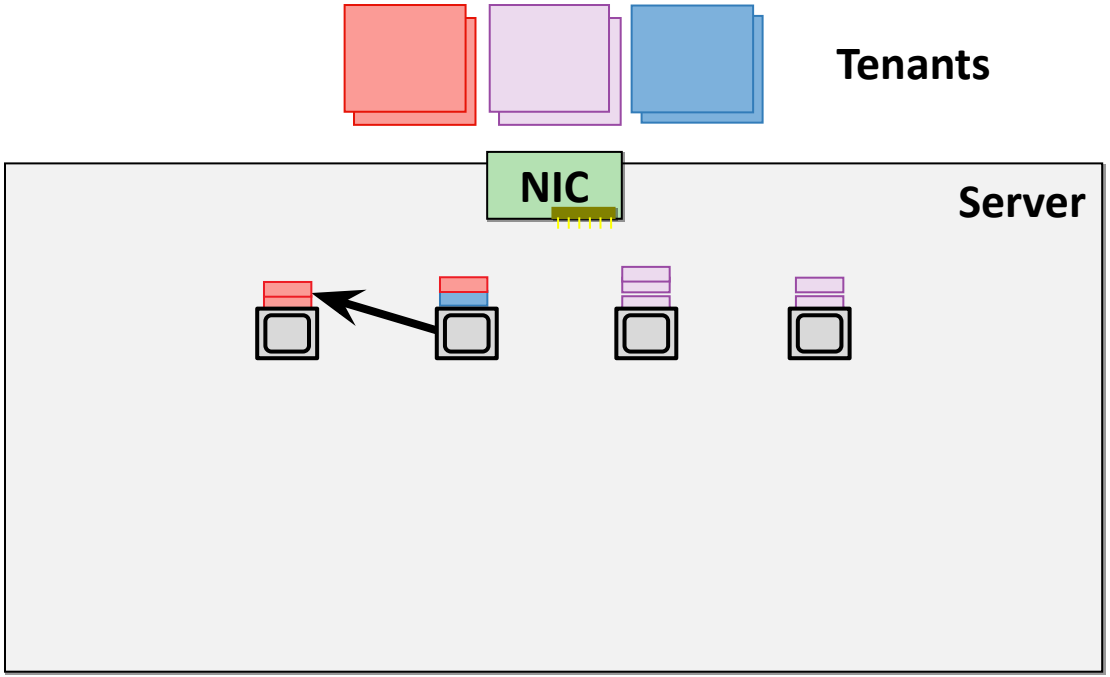


Few active tenants
Many idle tenants

Splinter: Tenant Locality And Work Stealing

Problem: **Quickly dispatch requests to cores, avoid hotspots**

Solution: **NIC routes tenants to cores, cores steal work**

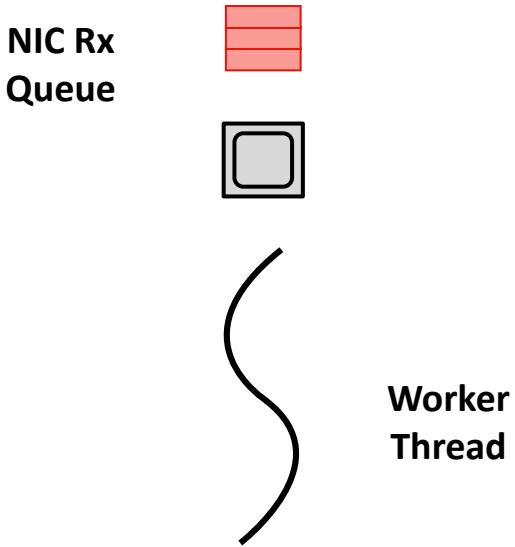


Cores steal from neighboring queue

Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

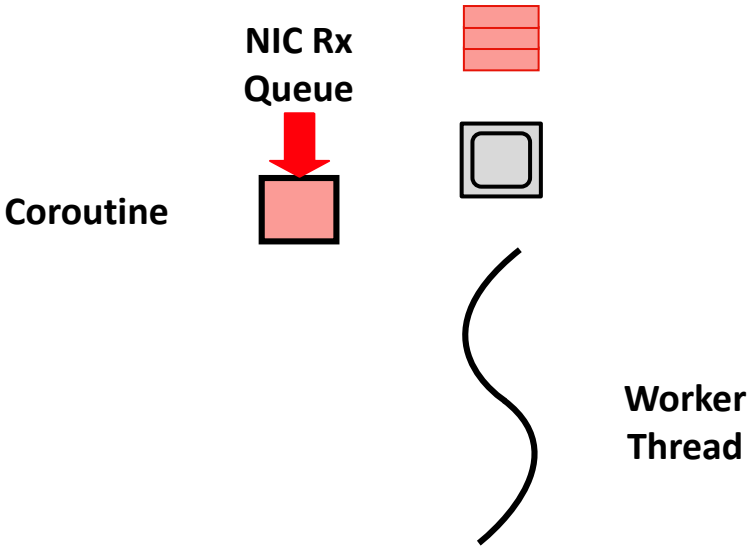
Solution: **Run extensions in stackless coroutines**



Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

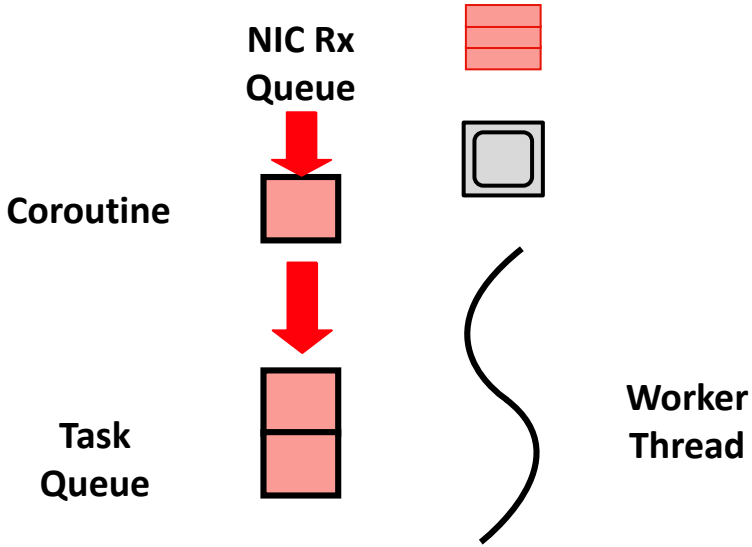
Solution: **Run extensions in stackless coroutines**



Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

Solution: **Run extensions in stackless coroutines**

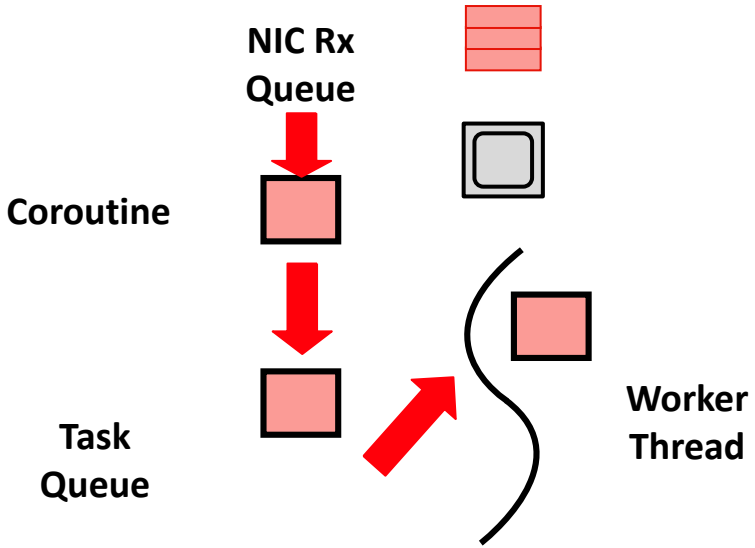


Dedicated Dispatch task to construct coroutines

Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

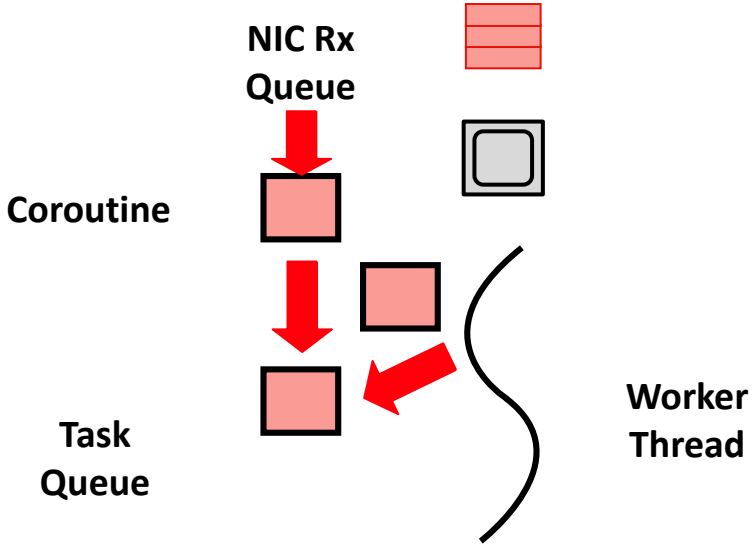
Solution: **Run extensions in stackless coroutines**



Splinter: Lightweight Cooperative Scheduling

Problem: **Minimize trust boundary crossing cost**

Solution: **Run extensions in stackless coroutines**



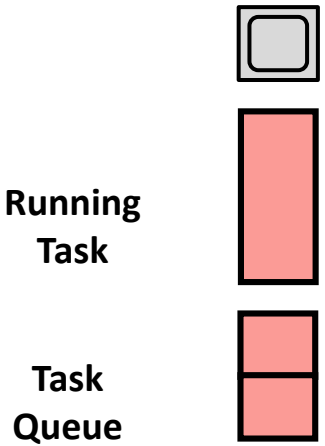
Task switch cost ~10ns

Run extension until it returns

Splinter: Lightweight Cooperative Scheduling

Problem: **Long running tasks starve shorter tasks, hurt latency**

Solution: **Extensions are cooperative, must yield frequently**

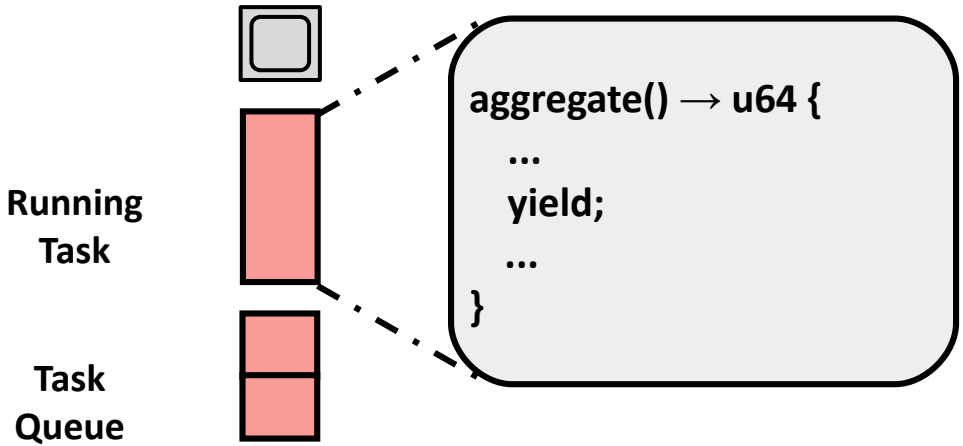


Preemption is too expensive!

Splinter: Lightweight Cooperative Scheduling

Problem: **Long running tasks starve shorter tasks, hurt latency**

Solution: **Extensions are cooperative, must yield frequently**

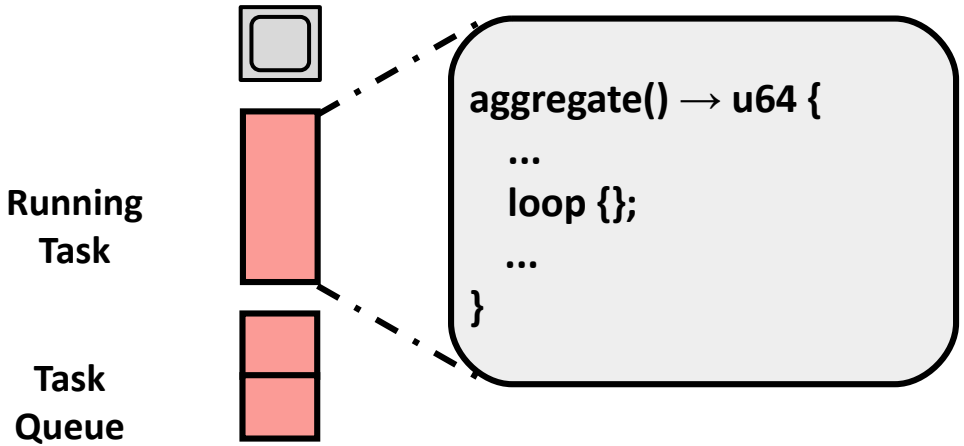


Compiler generates code to save & restore state

Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

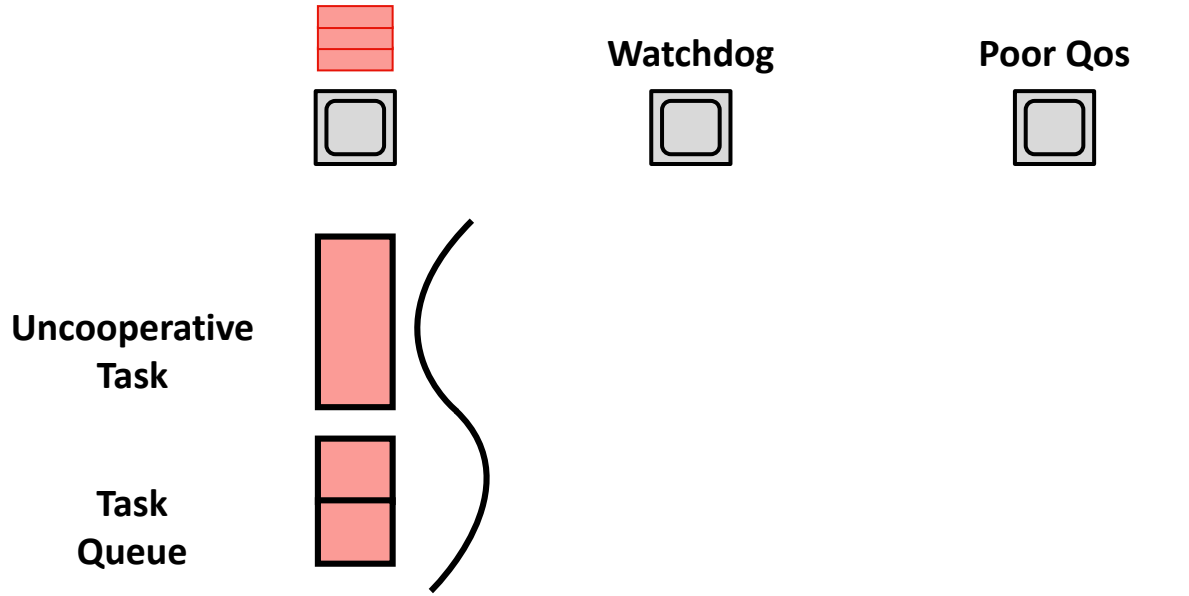
Solution: **Trusted watchdog core**



Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

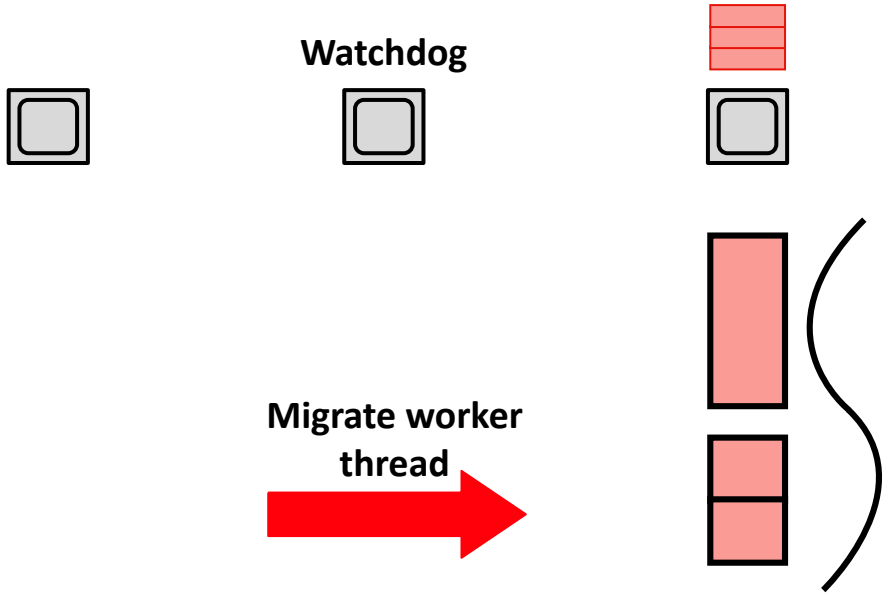
Solution: **Trusted watchdog core**



Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

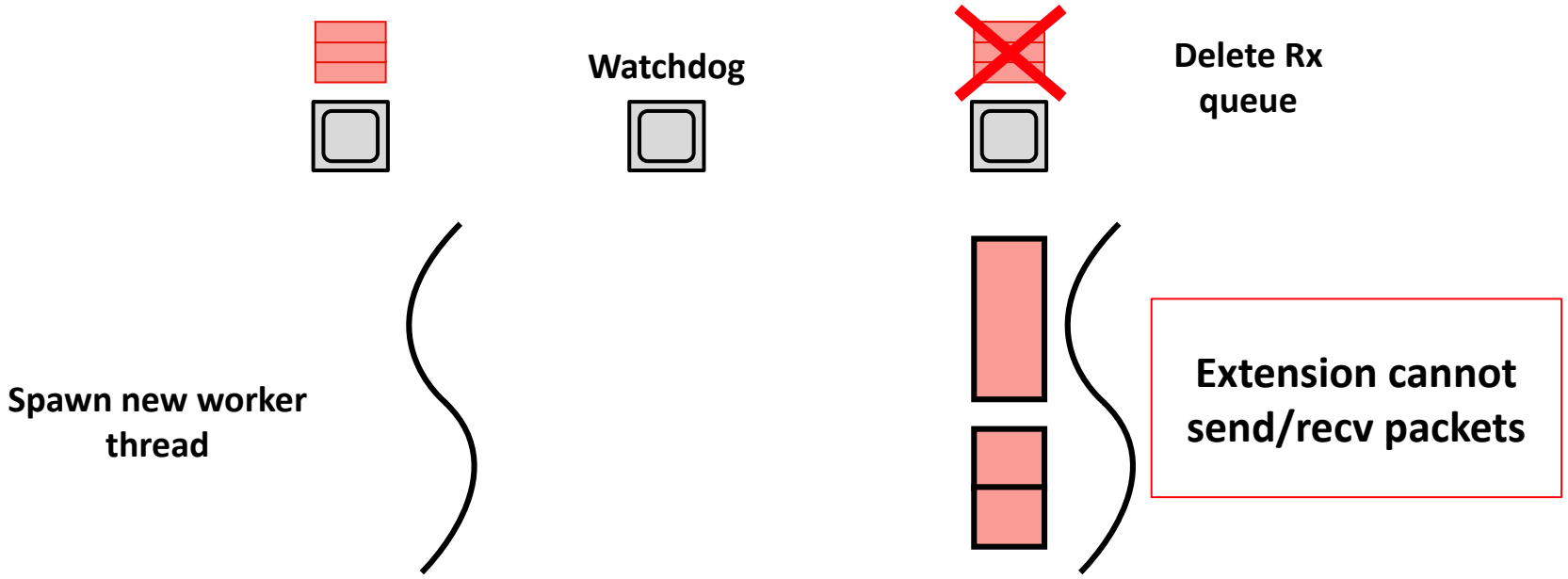
Solution: **Trusted watchdog core**



Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

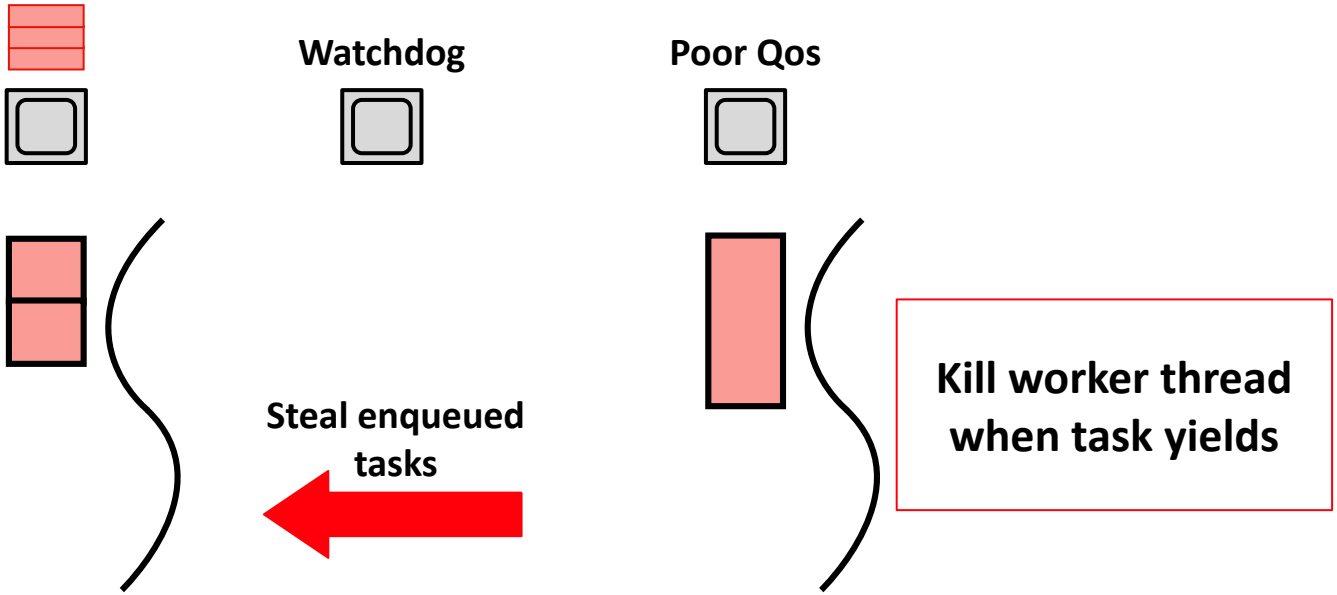
Solution: **Trusted watchdog core**



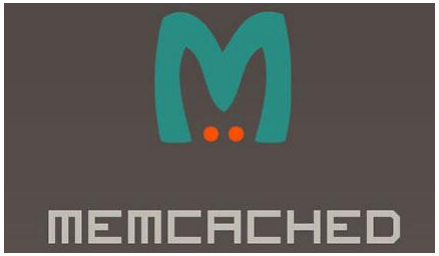
Splinter: Lightweight Cooperative Scheduling

Problem: **Uncooperative extensions**

Solution: **Trusted watchdog core**



Key-Value Stores are Used Everywhere!



Key-Value Stores are Used Everywhere!

MEMCACHED

redis

RocksDB

amazon web services

DynamoDB

Scaling Memcache at Facebook

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, Venkateshwaran Venkataramani

{rajeshn,hans}@fb.com, {sgrimm, marc}@facebook.com, {herman, hcli, rm, mpal, dpeek, ps, dstaff, ttung, veeve}@fb.com

Facebook Inc.

Abstract: `Memcached` is a well known, simple, in-memory caching solution. This paper describes how Facebook leverages `memcached` as a building block to construct and scale a distributed key-value store that supports the world's largest social network. Our system handles billions of requests per second and holds trillions of items to deliver a rich experience for over a bil-

however, web pages routinely fetch thousands of key-value pairs from `memcached` servers.

One of our goals is to present the important themes that emerge at different scales of our deployment. While qualities like performance, efficiency, fault-tolerance, and consistency are important at all scales, our experience indicates that at specific sizes some qualities re-

NSDI 2013

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

SOSP 2007

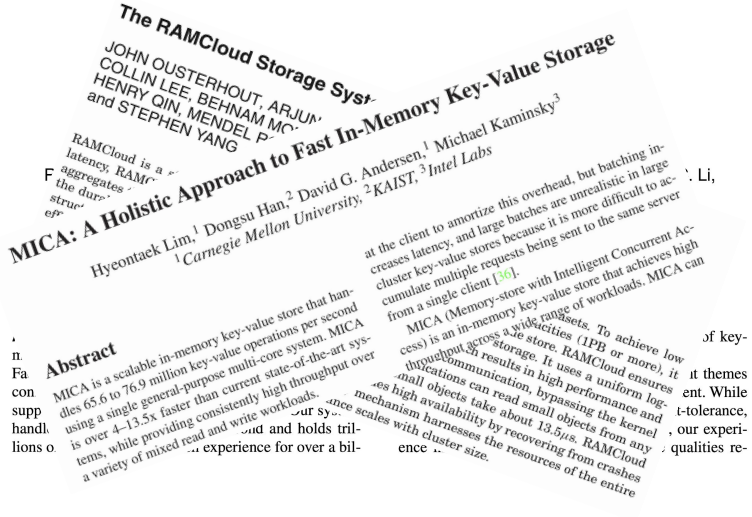
Key-Value Stores are Used Everywhere!

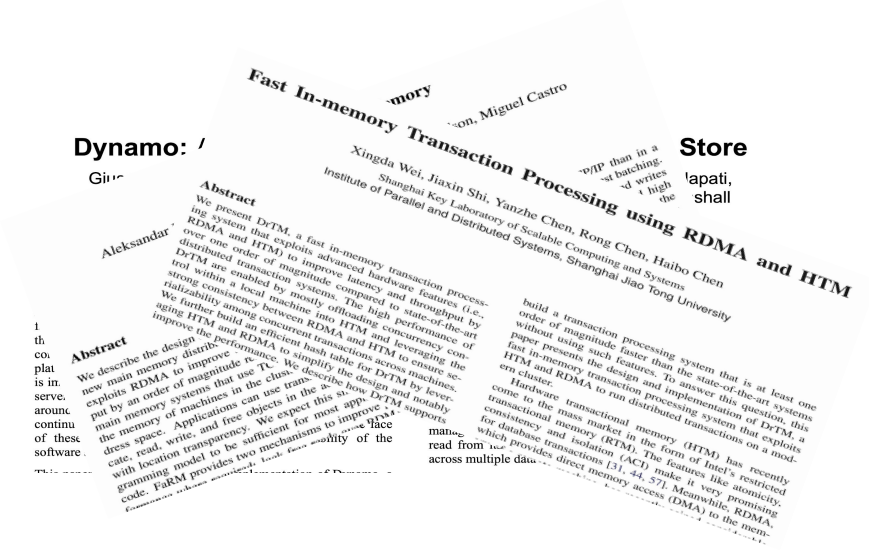
MEMCACHED

redis

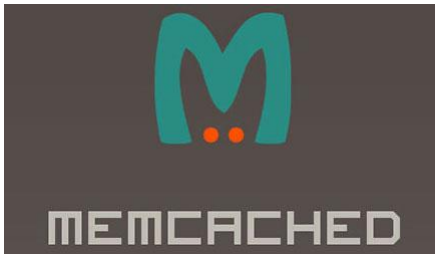
RocksDB

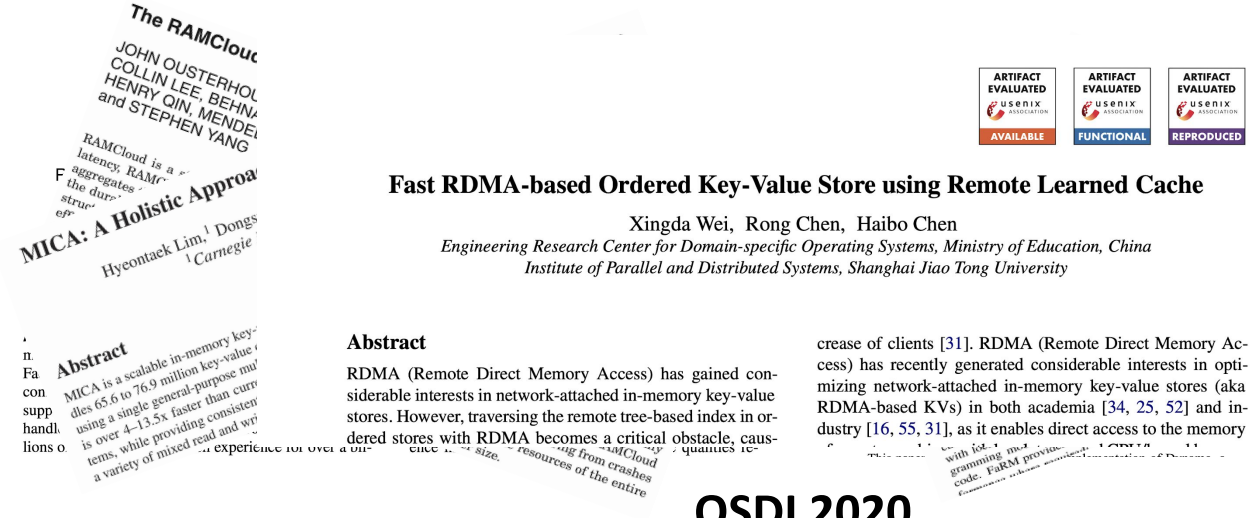
amazon web services
DynamoDB


The RAMCloud Storage Syst
JOHN OUSTERHOUT, ARJUN
COLLIN LEE, BEHNAM MO
HENRY QIN, MENDEL P
and STEPHEN YANG
MICA: A Holistic Approach to Fast In-Memory Key-Value Storage
Hyeontaek Lim,[1] Dongsu Han,[2] David G. Andersen,[1] Michael Kaminsky[3]
[1]Carnegie Mellon University, [2]KAIST, [3]Intel Labs
Abstract


Fast In-memory Transaction Processing using RDMA and HTM
Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, Haibo Chen
Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Abstract

Key-Value Stores are Used Everywhere!





OSDI 2020

Shadowfax: TODO

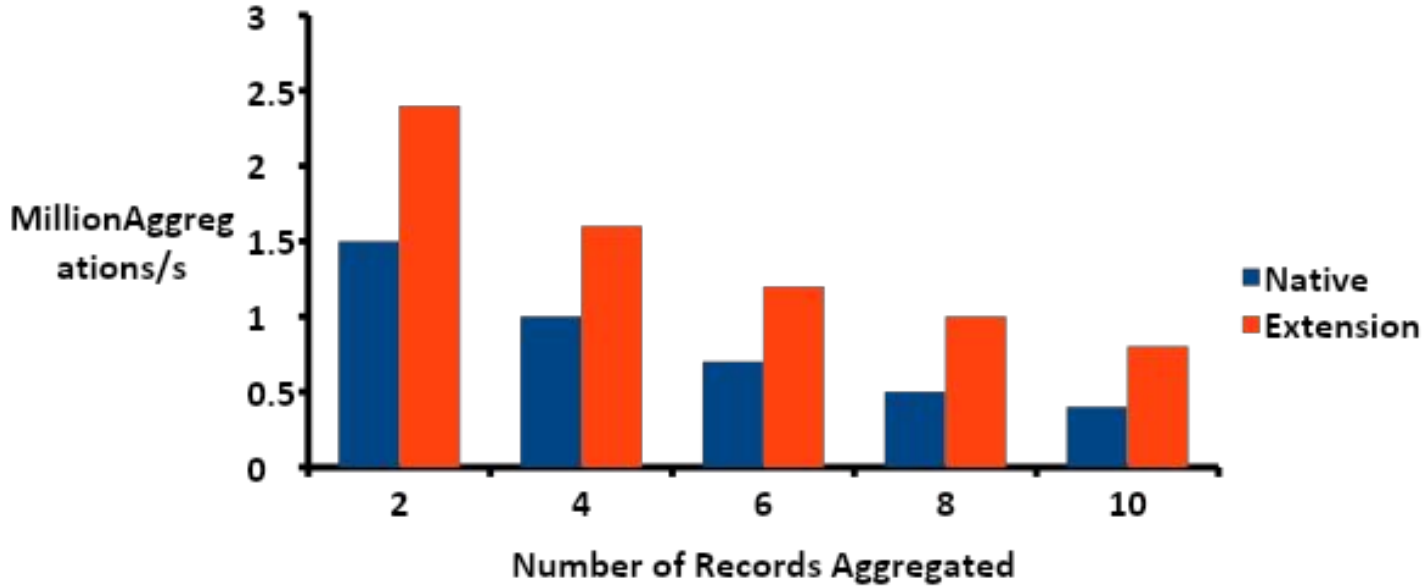
Analyze larger-than-memory aspect of system

- ➔ Throughput when data does not fit in main memory
- ➔ What limits performance? What should change to improve performance?
- ➔ **Badrish: Cost benefit analysis with larger-than-memory**

Thesis Timeline

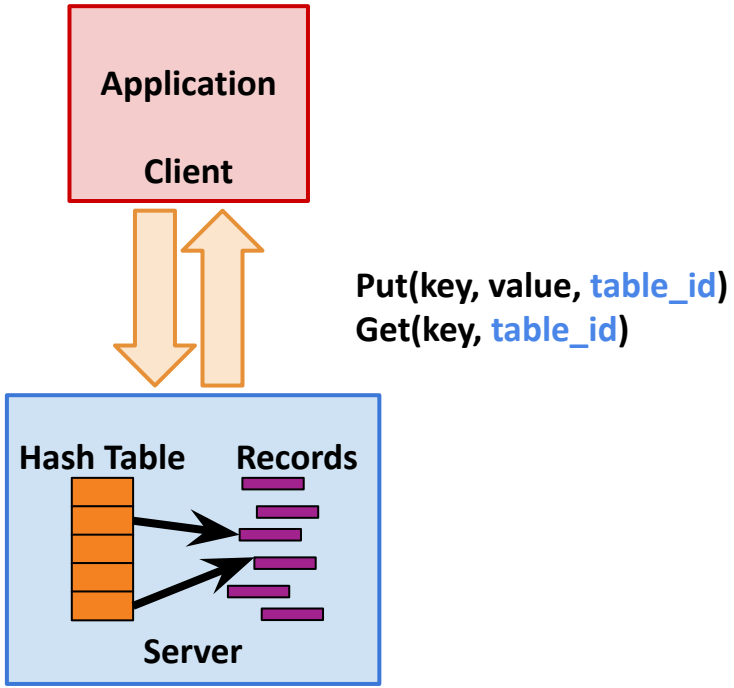
Task 1	Develop fast data migration protocol for low-latency stores	Feb 2017	Done
Task 2	Write up and submit results to SOSP 2017	Apr 2017	Done
Task 3	Develop mechanism for extensibility and multi-tenancy for low-latency stores	Mar 2018	Done
Task 4	Write up and submit results to OSDI 2018	May 2018	Done
Task 5	Develop mechanisms for low cost coordination in low-latency stores	Mar 2019	Done
Task 6	Develop mechanisms for scale-out with cloud storage in low-latency stores	Mar 2020	Done
Task 7	Write up and submit results to VLDB 2021	Sep 2020	Done
Task 8	Finish Writing Thesis Introduction	Nov 2020	Done
Task 9	Analyze normal-case store performance with cloud storage	Dec 2020	TBD
Task 10	Finish writing low cost coordination chapter	Jan 2021	TBD
Task 11	Finish writing Conclusion	Jan 2021	TBD
Task 12	Final thesis defense	Feb 2021	TBD

Simple Aggregation With Splinter



Extension Mode → Few RPCs, Less Data movement → Better Throughput

Key-Value Stores: Typical Deployment



Put(key, value, table_id)
Get(key, table_id)

Some systems support tables

64 bit hash space per table

Adaptive Placement for In-Memory Storage Functions

Ankit Bhardwaj (Labmate) has built on Splinter

➔ For certain functions, client-side execution (gets & puts) is better

ASFP: Adaptive Storage Function Placement

➔ Decouples function placement from execution

➔ Dynamically models and decides where to execute functions

➔ Published at ATC'20 (I am the second author)