Operating Systems (Honor Track)

Scheduling 4: Deadlock & Scheduling in Modern Computer Systems

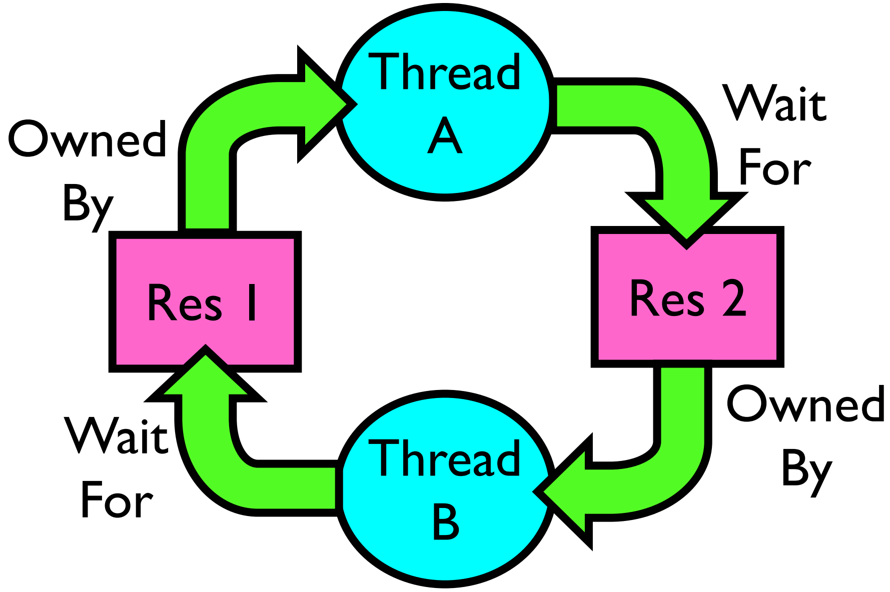
Xin Jin

Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Recap: Deadlock: A Deadly type of Starvation

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
    Thread B owns Res 2 and is waiting for Res 1
- Deadlock $\Rightarrow$ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention



Recap: Four requirements for occurrence of Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

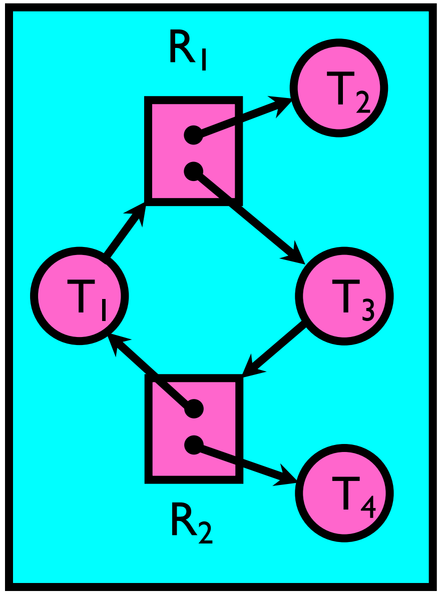
Recap: Deadlock Detection Algorithm

- Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

`[FreeResources]:`	Current free resources each type
$[Request_X]$:	Current requests from thread X
$[Alloc_X]$:	Current resources held by thread X

- See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   For each node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



- Nodes left in UNFINISHED $\Rightarrow$ deadlocked

How should a system deal with deadlock?

- Four different approaches:

1. Deadlock prevention: write your code in a way that it isn't prone to deadlock
2. Deadlock recovery: let deadlock happen, and then figure out how to recover from it
3. Deadlock avoidance: dynamically delay resource requests so deadlock doesn't happen
4. Deadlock denial: ignore the possibility of deadlock

- Modern operating systems:
  - Make sure the *system* isn't involved in any deadlock
  - Ignore deadlock in applications
    - "Ostrich Algorithm"

Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call Mom in Toledo, works way through phone network, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - » Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to retry!

(Virtually) Infinite Resources

Thread A	**Thread B**
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`AllocateOrWait(1 MB)`	`AllocateOrWait(1 MB)`
`Free(1 MB)`	`Free(1 MB)`
`Free(1 MB)`	`Free(1 MB)`

- With virtual memory we have "infinite" space so everything will just succeed, thus above example won't deadlock
  - Of course, it isn't actually infinite, but certainly larger than 2MB!

Techniques for Preventing Deadlock

- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (`x.Acquire()`, `y.Acquire()`, `z.Acquire()`,…)
    - » Make tasks request disk, then memory, then…
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Request Resources Atomically (1)

Rather than:

Thread A:

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

Consider instead:

Thread A:

```
Acquire_both(x, y);
…
y.Release();
x.Release();
```

Thread B:

```
Acquire_both(y, x);
…
x.Release();
y.Release();
```

Request Resources Atomically (2)

Or consider this:

Thread A

```
z.Acquire();
x.Acquire();
y.Acquire();
z.Release();
…
y.Release();
x.Release();
```

Thread B

```
z.Acquire();
y.Acquire();
x.Acquire();
z.Release();
…
x.Release();
y.Release();
```

Acquire Resources in Consistent Order

Rather than:

Thread A:
```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:
```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

Consider instead:

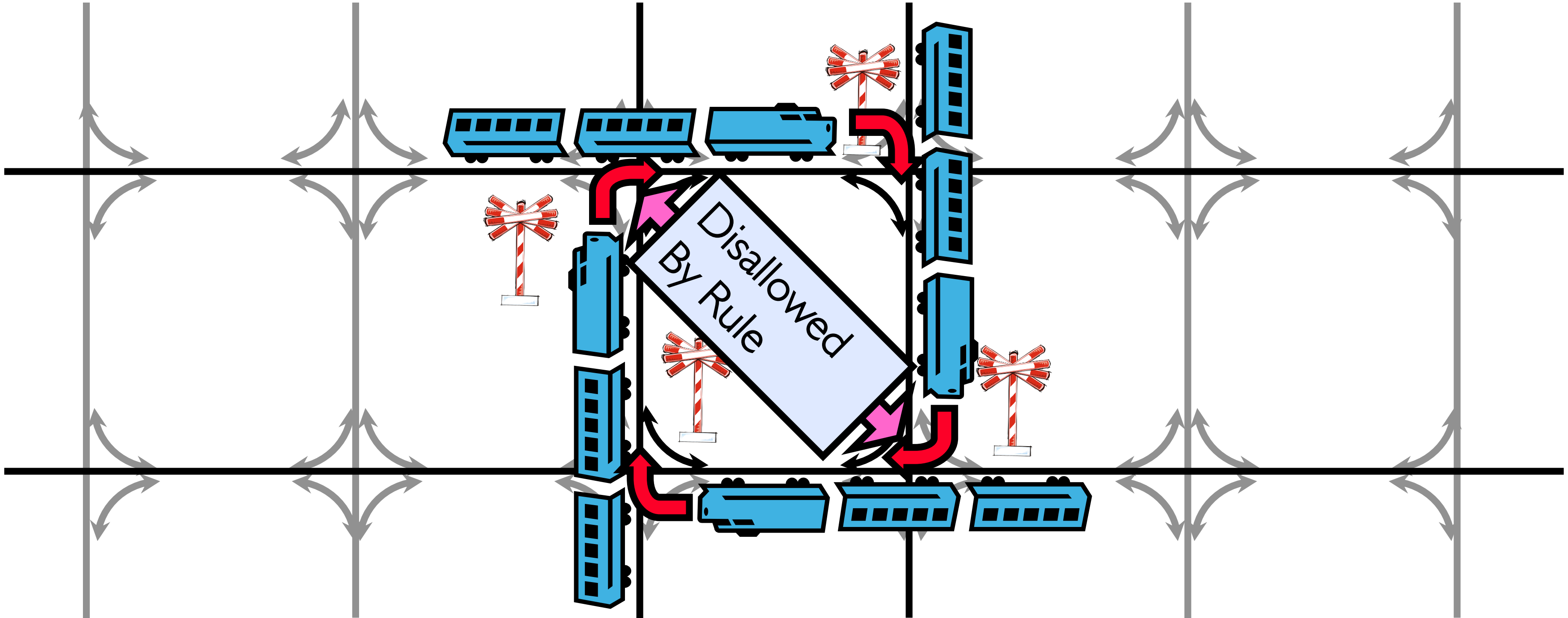
Thread A:
```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:
```
x.Acquire();
y.Acquire();
…
x.Release();
y.Release();
```

Does it matter in which order the locks are released?

Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Techniques for Recovering from Deadlock

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Hold dining lawyer in contempt and take away in handcuffs
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn’t always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Another view of virtual memory: Pre-empting Resources

<u>Thread A</u>:

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

<u>Thread B</u>:

```
AllocateOrWait(1 MB)
AllocateOrWait(1 MB)
Free(1 MB)
Free(1 MB)
```

- Before: With virtual memory we have "infinite" space so everything will just succeed, thus above example won't deadlock
  - Of course, it isn't actually infinite, but certainly larger than 2MB!
- Alternative view: we are "pre-empting" memory when paging out to disk, and giving it back when paging back in
  - This works because thread can't use memory when paged out

Techniques for Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in deadlock
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

THIS DOES NOT WORK!!!!

- Example:

Thread A:

```
x.Acquire();
y.Acquire();        Blocks…
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();
x.Acquire();        Wait?
…                   But it's already too late…
x.Release();
y.Release();
```

Deadlock Avoidance: Three States

- Safe state
  - System can delay resource acquisition to prevent deadlock
- Unsafe state
  - No deadlock yet…
  - But threads can request resources in a pattern that ***unavoidably*** leads to deadlock

Deadlock avoidance: prevent system from reaching an *unsafe* state

- Deadlocked state
  - There exists a deadlock in the system
  - Also considered "unsafe"

Deadlock Avoidance

- Idea: When a thread requests a resource, OS checks if it would result in ~~deadlock~~ an unsafe state
  - If not, it grants the resource right away
  - If so, it waits for other threads to release resources

- Example:

Thread A:

```
x.Acquire();
y.Acquire();
…
y.Release();
x.Release();
```

Thread B:

```
y.Acquire();
x.Acquire();
…
x.Release();
y.Release();
```

Wait until Thread A releases mutex X

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) ≥ max
    remaining that might be needed by any thread

- Banker's algorithm:
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

      $([Max_{node}]-[Alloc_{node}] <= [Avail])$ for $([Request_{node}] <= [Avail])$

      Grant request if result is deadlock free

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
     Add all nodes to UNFINISHED
     do {
        done = true
        For each node in UNFINISHED {
           if ([Request_node] <= [Avail]) {
              remove node from UNFINISHED
              [Avail] = [Avail] + [Alloc_node]
              done = false
           }
        }
     } until(done)
```

- » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

  $([Max_{node}]-[Alloc_{node}] <= [Avail])$ for $([Request_{node}] <= [Avail])$

  Grant request if result is deadlock free

Banker's Algorithm for Avoiding Deadlock

```
[Avail] = [FreeResources]
     Add all nodes to UNFINISHED
     do {
        done = true
        For each node in UNFINISHED {
           if ([Max_node]-[Alloc_node] <= [Avail]) {
              remove node from UNFINISHED
              [Avail] = [Avail] + [Alloc_node]
              done = false
           }
        }
     } until(done)
```

- » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

  ([Max_{node}]-[$Alloc_{node}$] <= [Avail]) for ([$Request_{node}$] <= [Avail])

  Grant request if result is deadlock free

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) $\geq$ max remaining that might be needed by any thread

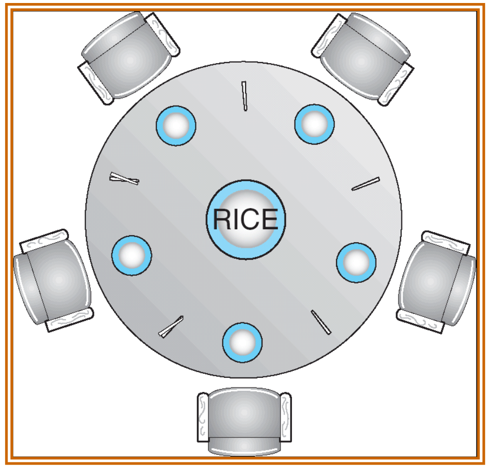
- Banker's algorithm:
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:

      $([Max_{node}]-[Alloc_{node}] <= [Avail])$ for $([Request_{node}] <= [Avail])$

      Grant request if result is deadlock free
  - Keeps system in a "SAFE" state: there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

Banker's Algorithm Example

- Banker's algorithm with dining lawyers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards

  - What if k-handed lawyers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Summary

- Four conditions for deadlocks
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Techniques for addressing Deadlock
  - Deadlock prevention:
    - » write your code in a way that it isn't prone to deadlock
  - Deadlock recovery:
    - » let deadlock happen, and then figure out how to recover from it
  - Deadlock avoidance:
    - » dynamically delay resource requests so deadlock doesn't happen
    - » Banker's Algorithm provides on algorithmic way to do this
  - Deadlock denial:
    - » ignore the possibility of deadlock

Scheduling in Modern Computer Systems

- FCFS
  – SOSP’17 ZygOS
- RR
  – NSDI’19 Shinjuku
- MLFQ
  – NSDI’19 Tiresias
- Fairness
  – NSDI’11 DRF
  – NSDI’16 FairRide

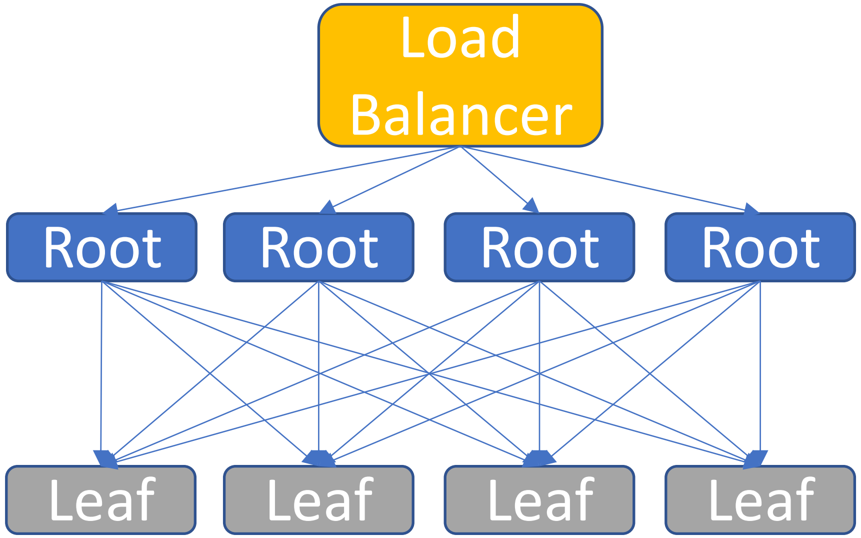
ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks

George Prekas, **Marios Kogias**, Edouard Bugnion

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

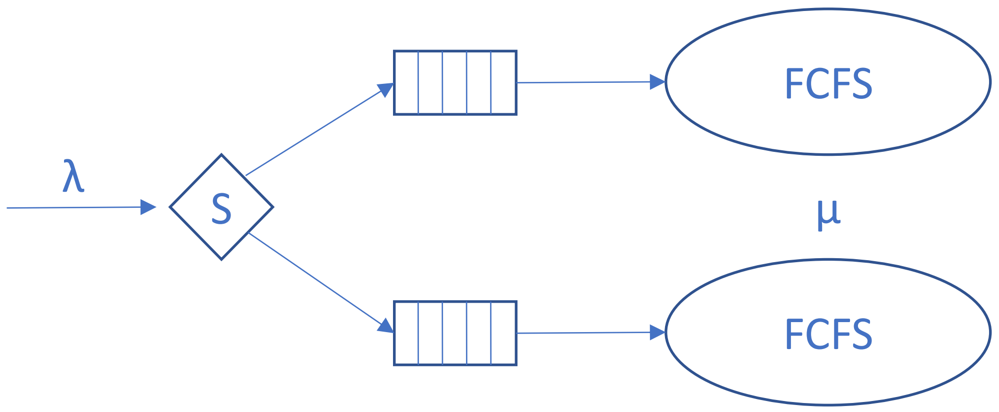
Problem: Serve μs-scale RPCs

- Applications: KV-stores, In-memory DB
- Datacenter environment:
  - Complex fan-out – fan-in patterns
- Tail-at-scale problem
- Tail Latency Service-Level Objectives
- Goal: Improve throughput at an aggressive tail latency SLO
- How? Focus within the leaf nodes
  - Reduce system overheads
  - Achieve better scheduling



Elementary Queuing Theory

- Processor
  - FCFS
  - Processor Sharing
- Multi/Single Queue
- Inter-arrival Distribution (λ)
  - Poisson
- Service Time Distribution (μ)
  - Fixed
  - Exponential
  - Bimodal



- No OS overheads
- Independent of service time
- Upper performance bound

Baseline

System	Linux		Dataplanes
Networking	Kernel (epoll)	Kernel (epoll)	Userspace
Connection Delegation	Partitioned	Floating	Partitioned
Complexity	Medium	High	Low
Work Conservation	✗	✓	✗
Queuing	Multi-Queue	Single Queue	Multi-Queue

Can we build a system with low overheads that achieves work conservation?

Upcoming

- Key Observations:
  - Single queue systems perform **theoretically** better
  - Dataplanes, despite being multi-queue systems, perform **practically** better

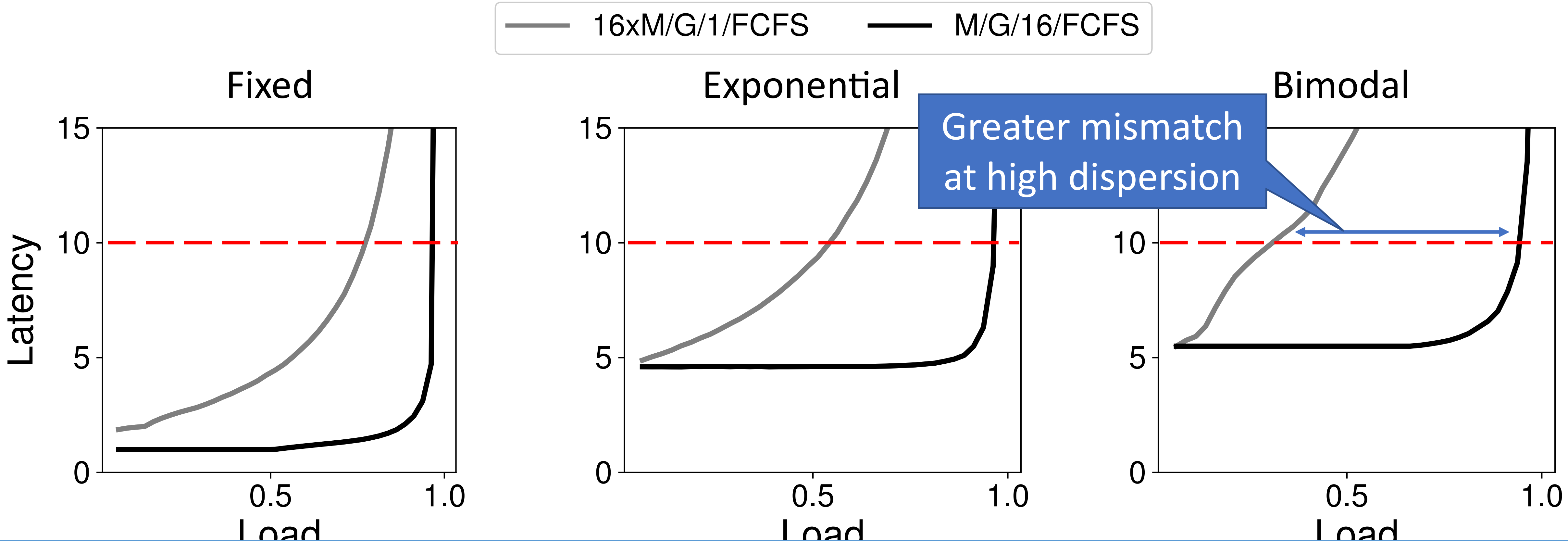
- Key Contributions
  - ZygOS combines the best of the two worlds:
    - Reduced system overheads similar to dataplanes
    - Convergence to a single-queue model

Analysis

- Metric to optimize: Load @ Tail-Latency SLO
- Run timescale-independent simulations
- Run synthetic benchmarks on real system

- Questions:
  - Which model achieves better throughput?
  - Which system converges to its model at low service times?

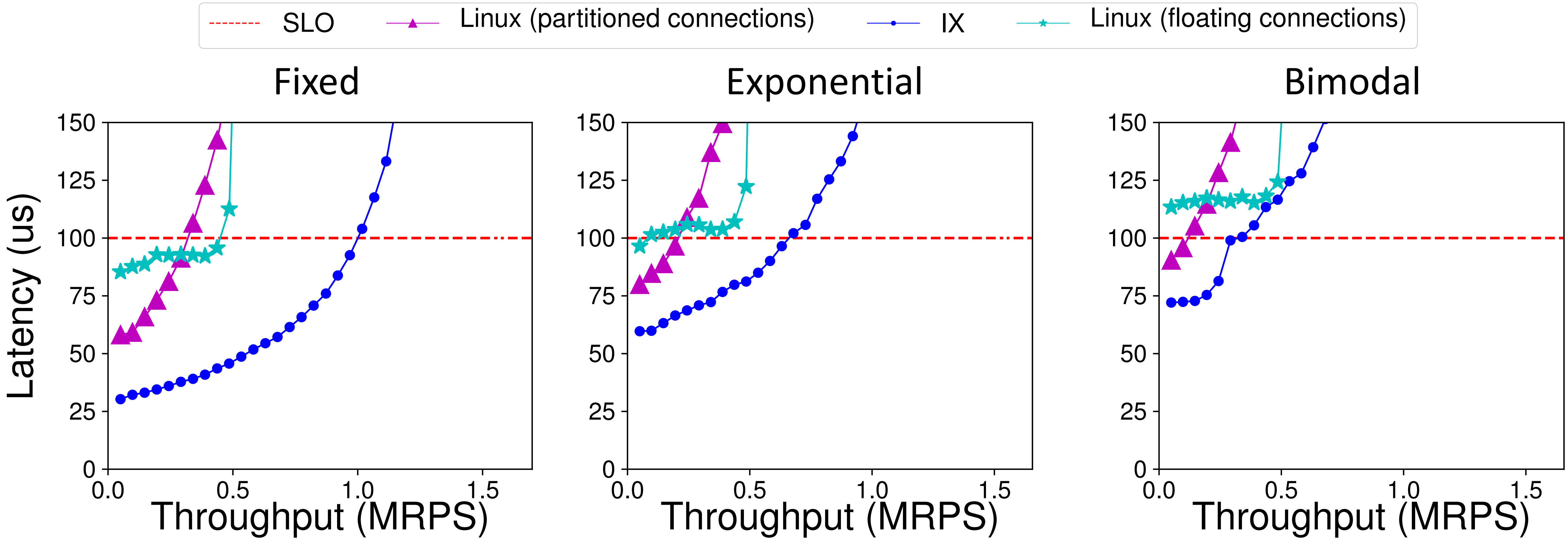
Latency vs Load – Queuing model



Single queue models provide better throughput at SLO because of **transient load imbalance**

SLO: 10 x AVG[service_time]

Latency vs Load – Service Time 10µs

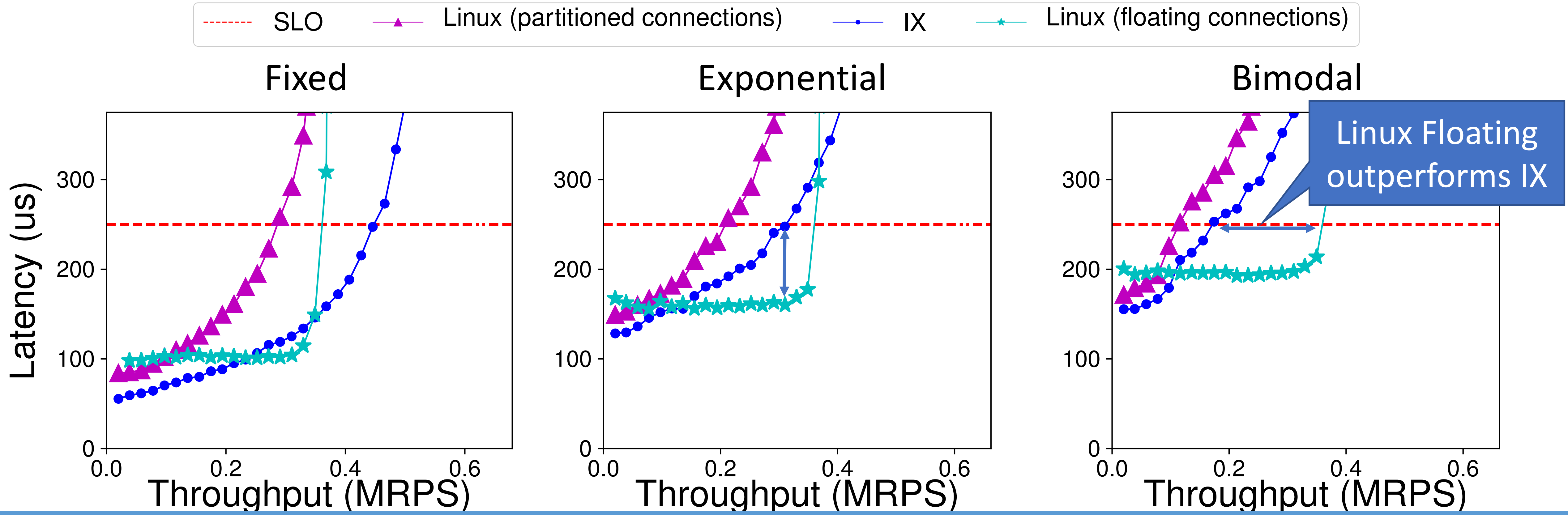


99th percentile latency

SLO: 10 x AVG[service_time]

IX, Belay et al. OSDI 2014

Latency vs Load – Service Time 25µs



Dataplanes perform better **only** in very low service times with low dispersion

99th percentile latency

SLO: 10 x AVG[service_time]

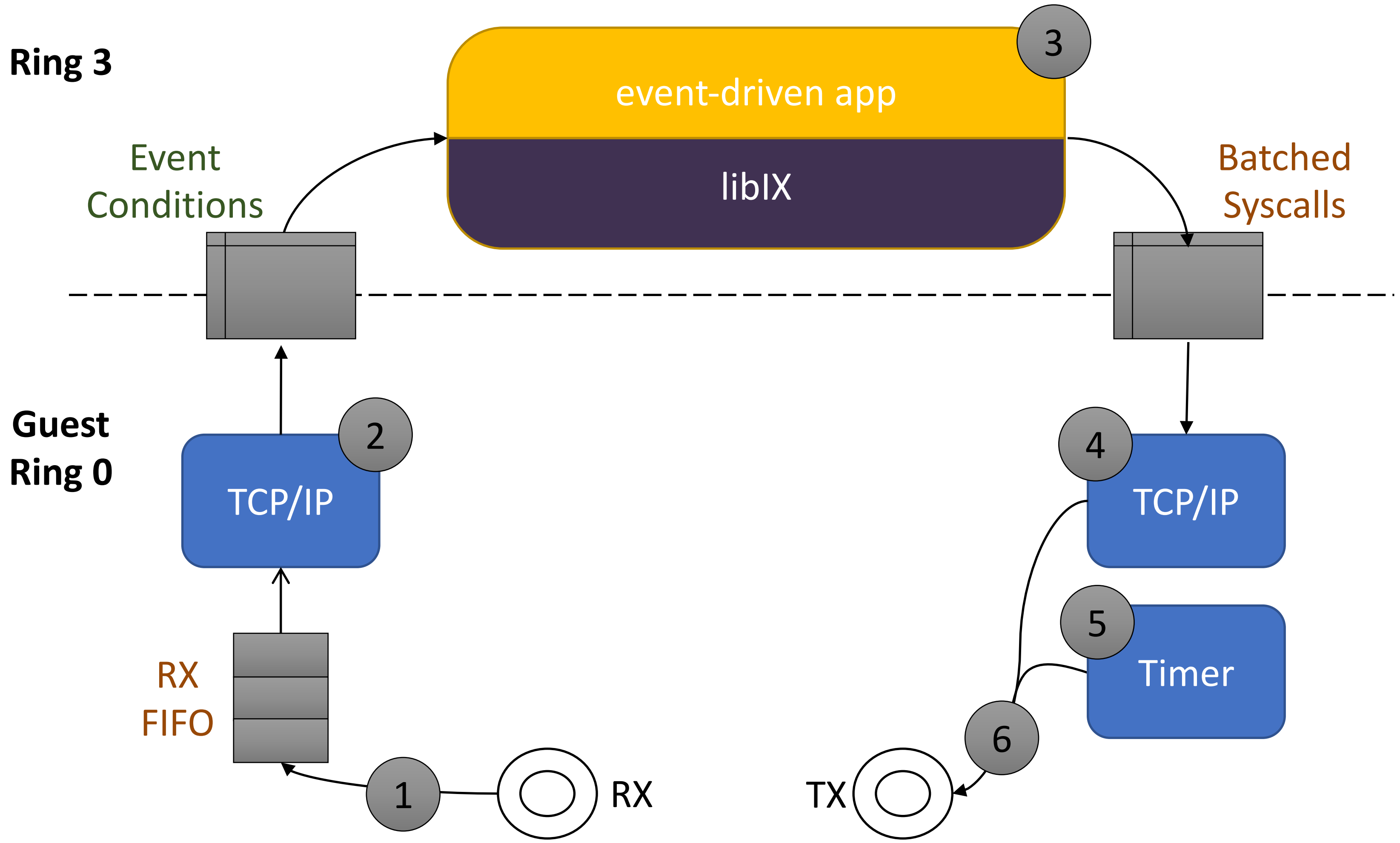
IX, Belay et al. OSDI 2014

ZygOS Approach

- Dataplane aspect:
  - Reduced system overheads
  - Share nothing network processing
- Single Queue system
  - Work conservation
  - Reduction of head of line blocking

Implement **work-stealing** to achieve work-conservation in a dataplane

Background on IX



IX Design

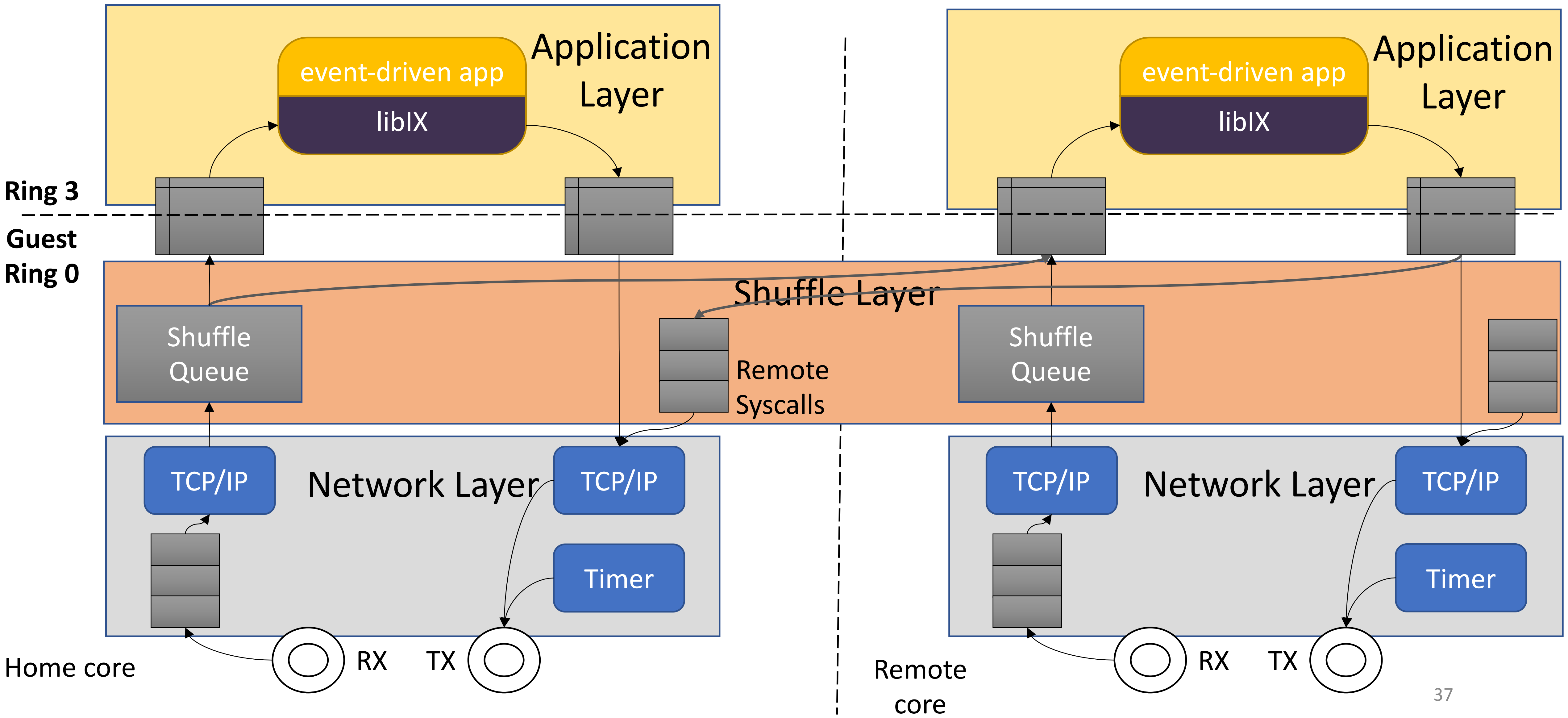
ZygOS Design

1. Application layer
   Event based application
   that is agnostic to work-stealing

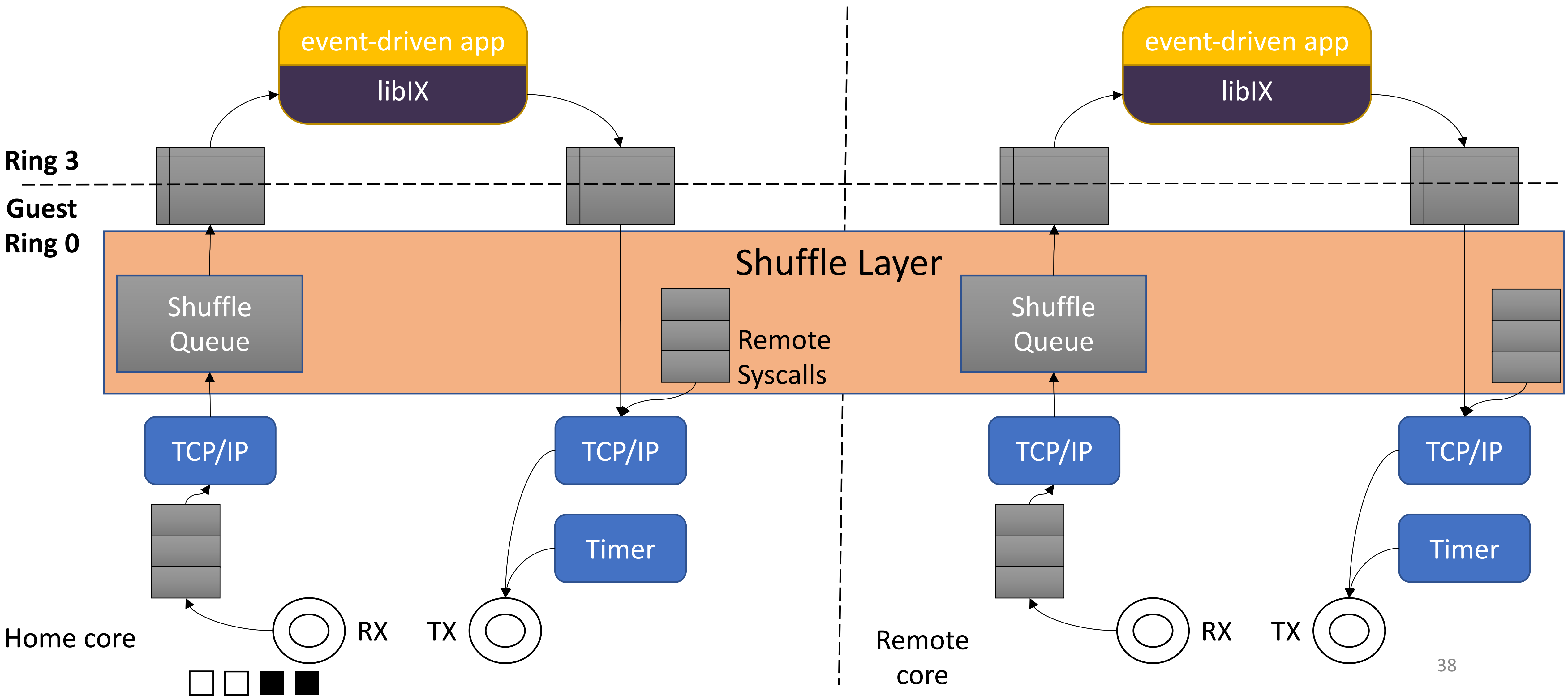
2. Shuffle layer
   Includes a per core list of ready connections that allows stealing

3. Network layer
   Coherence- and sync-free network processing

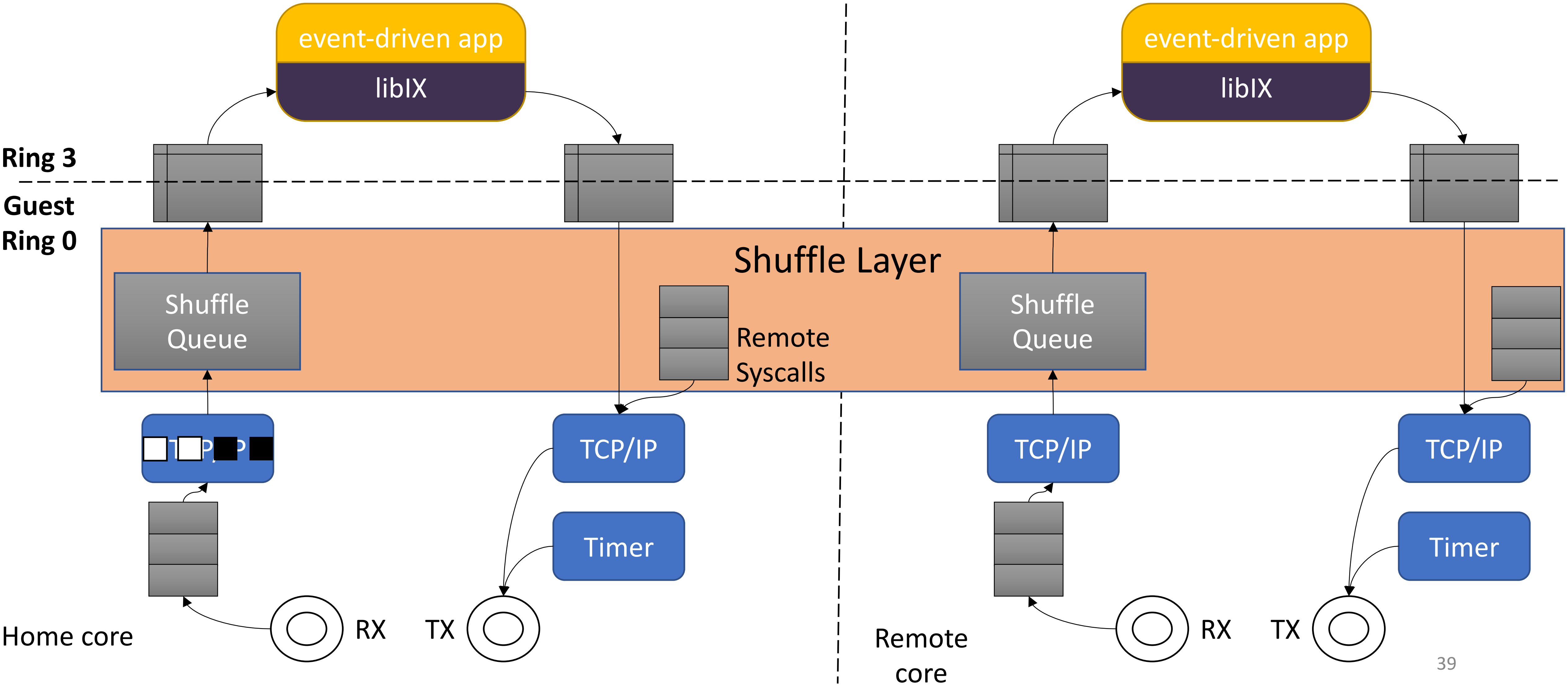
ZygOS Architecture



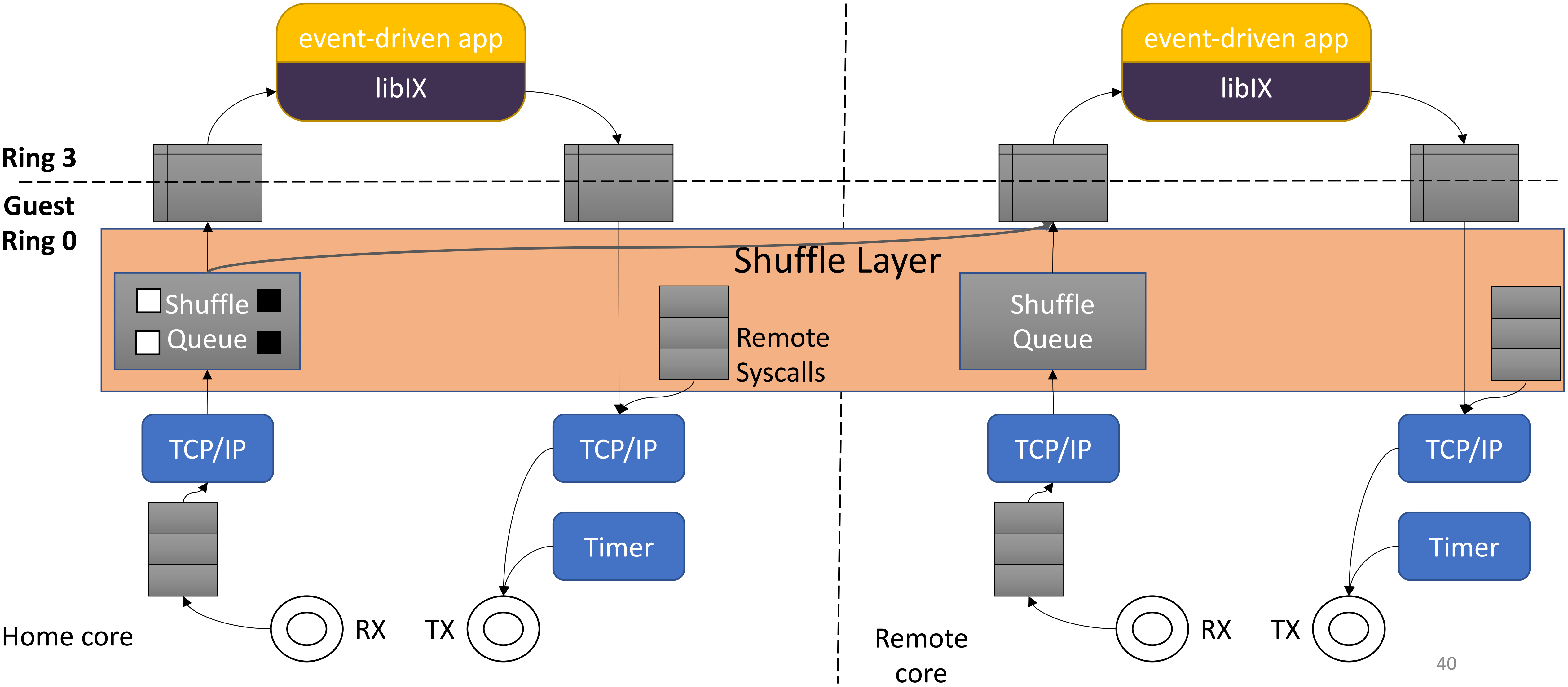
Execution Model



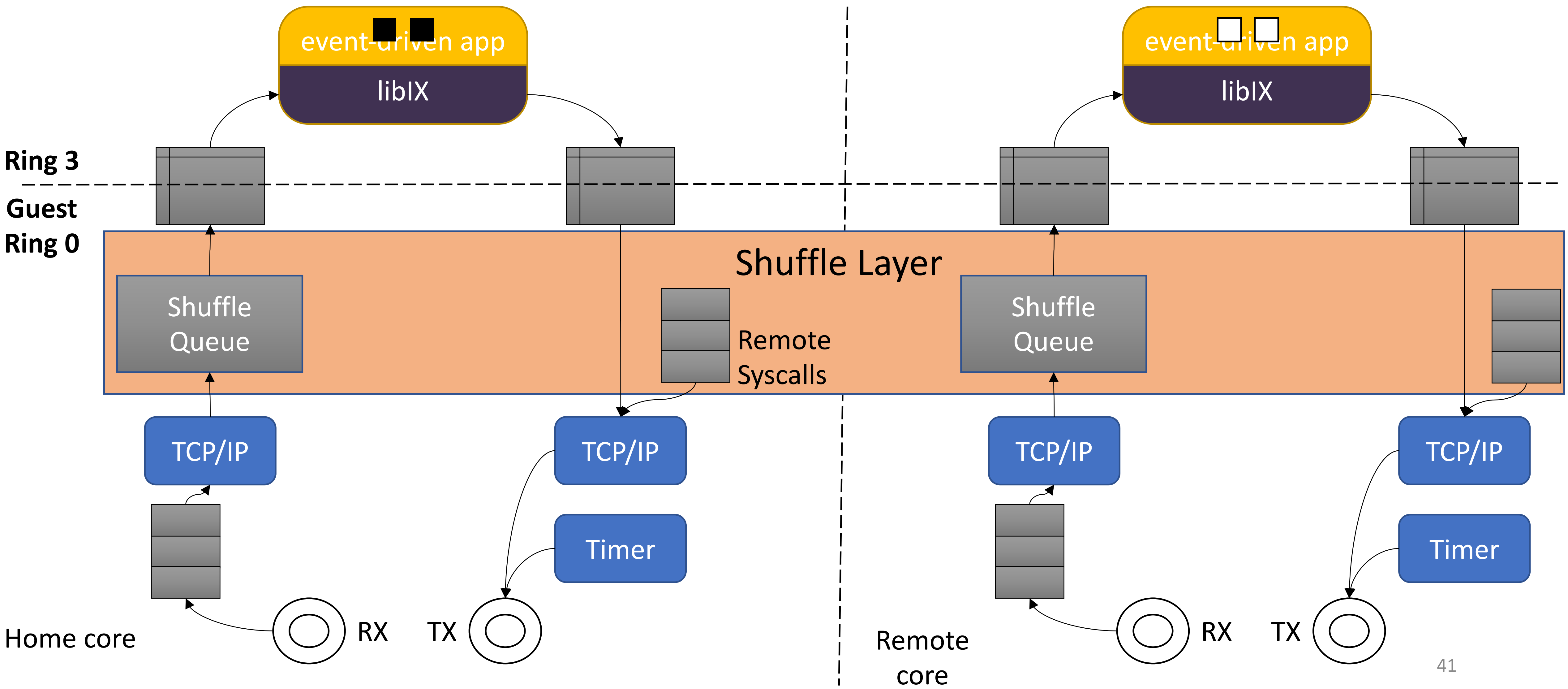
Execution Model



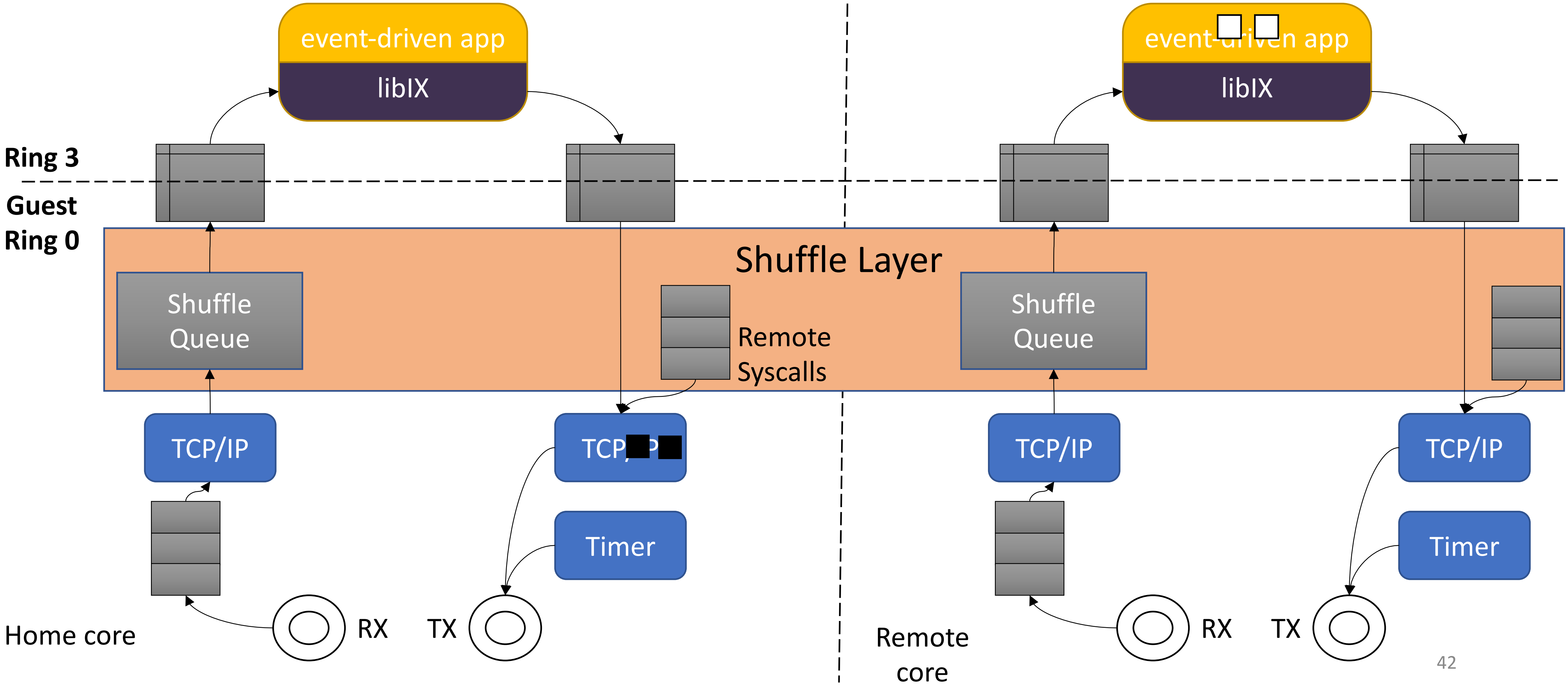
Execution Model



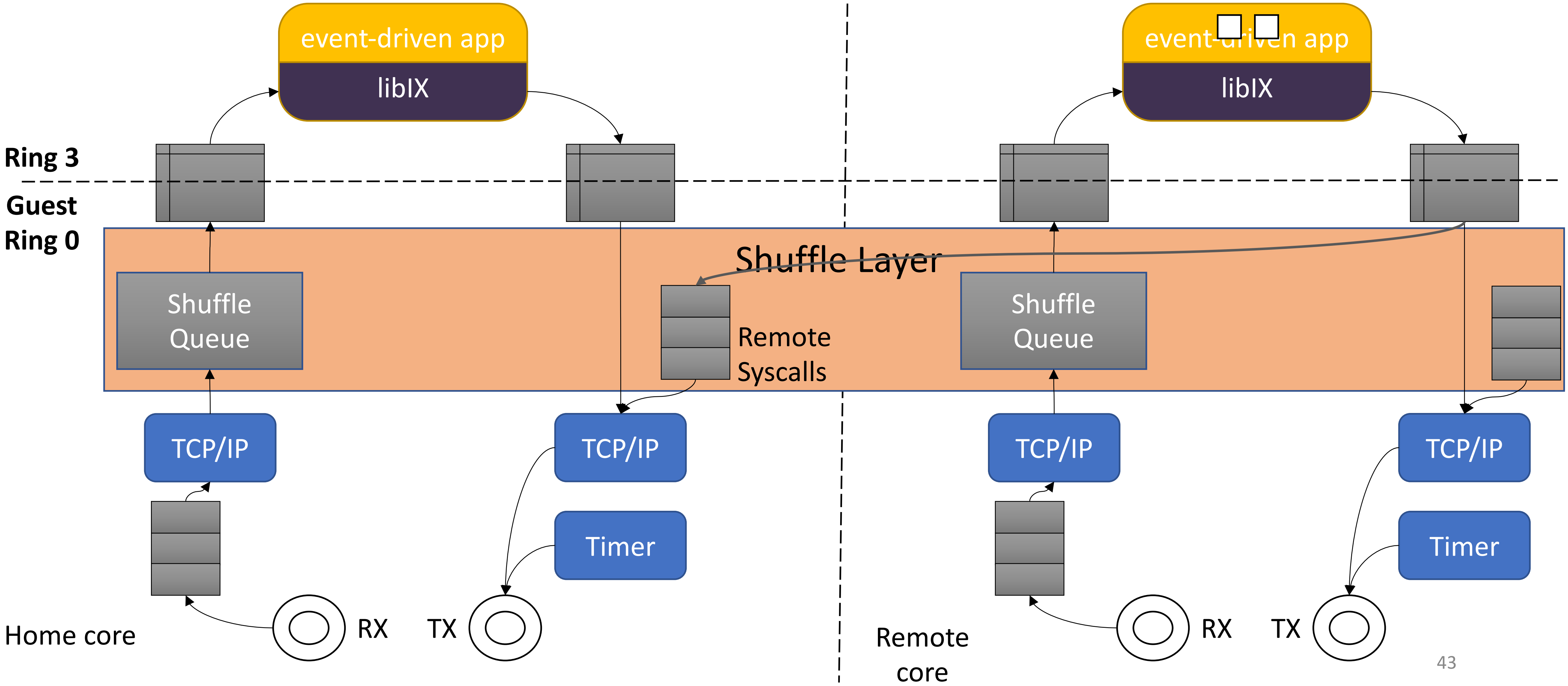
Execution Model



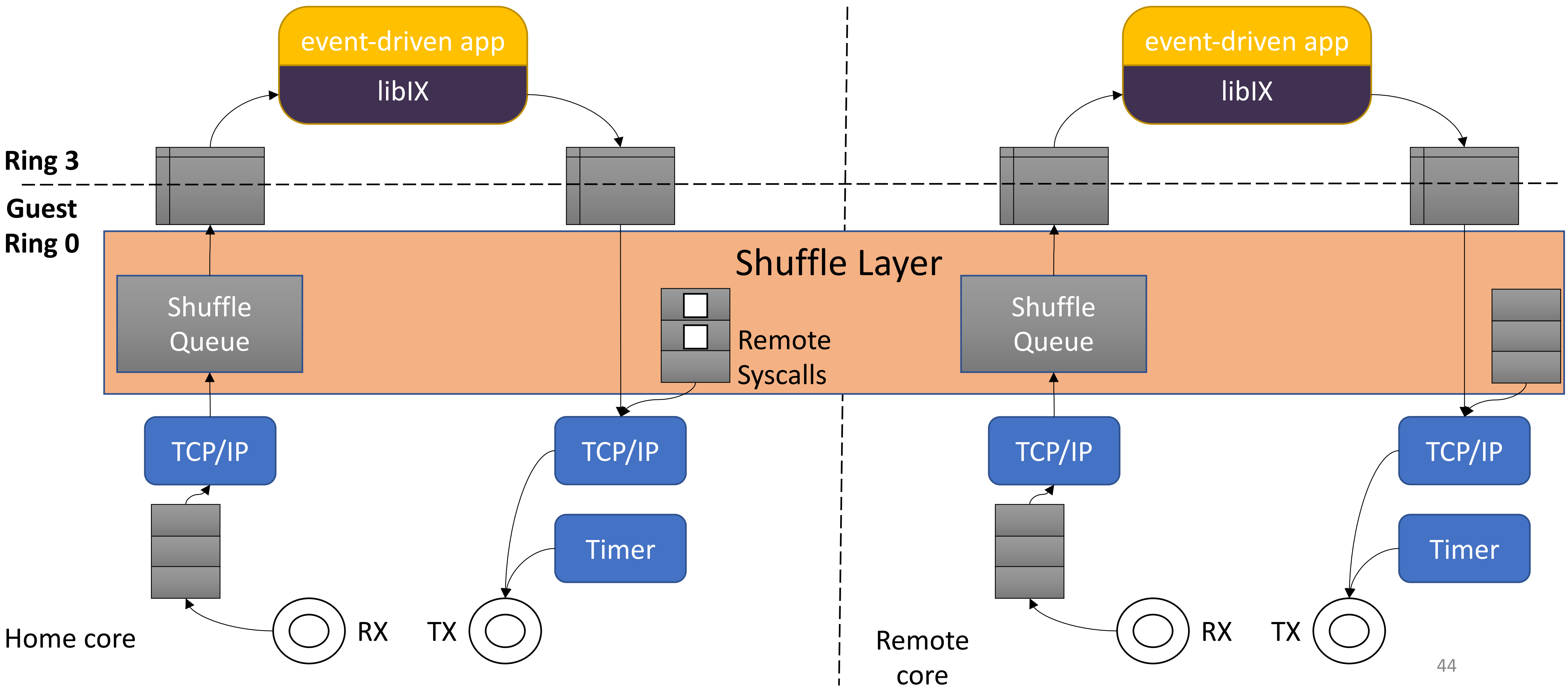
Execution Model



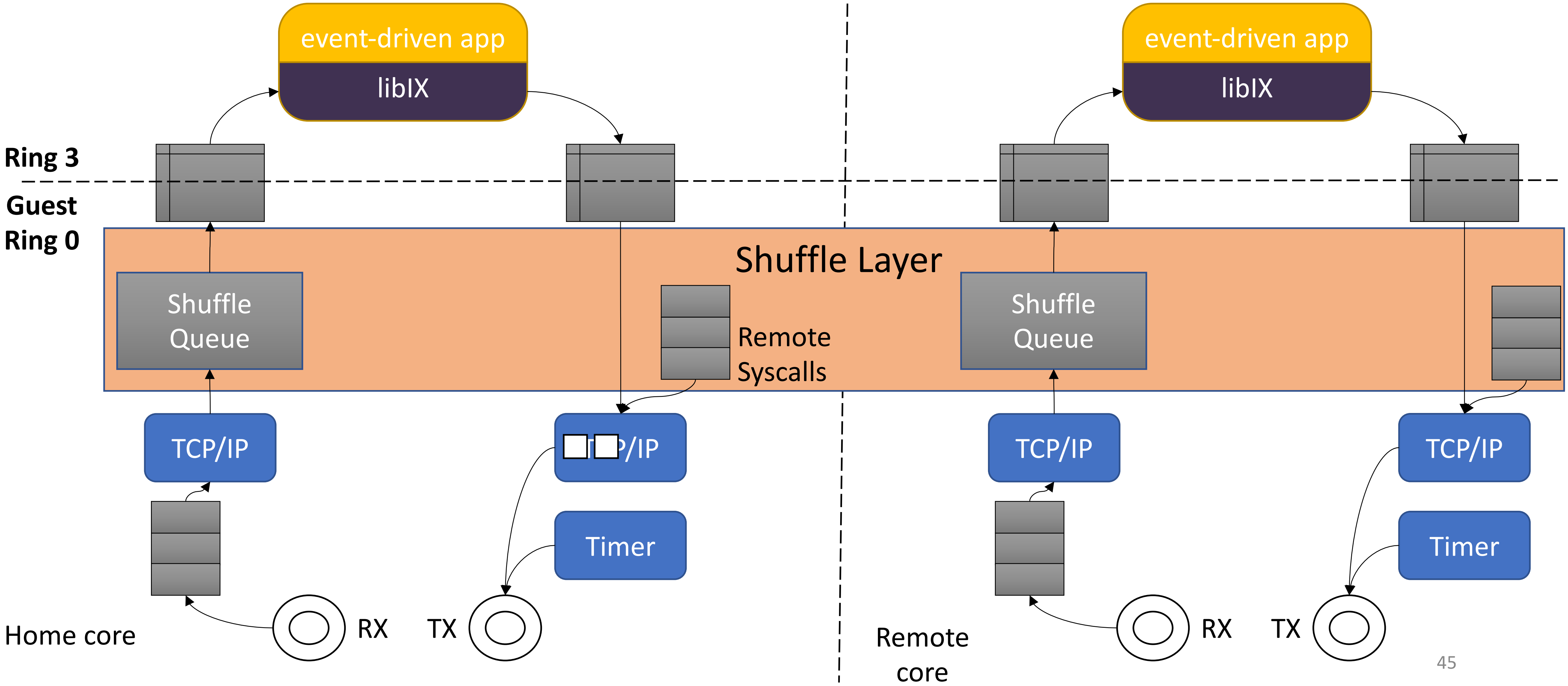
Execution Model



Execution Model



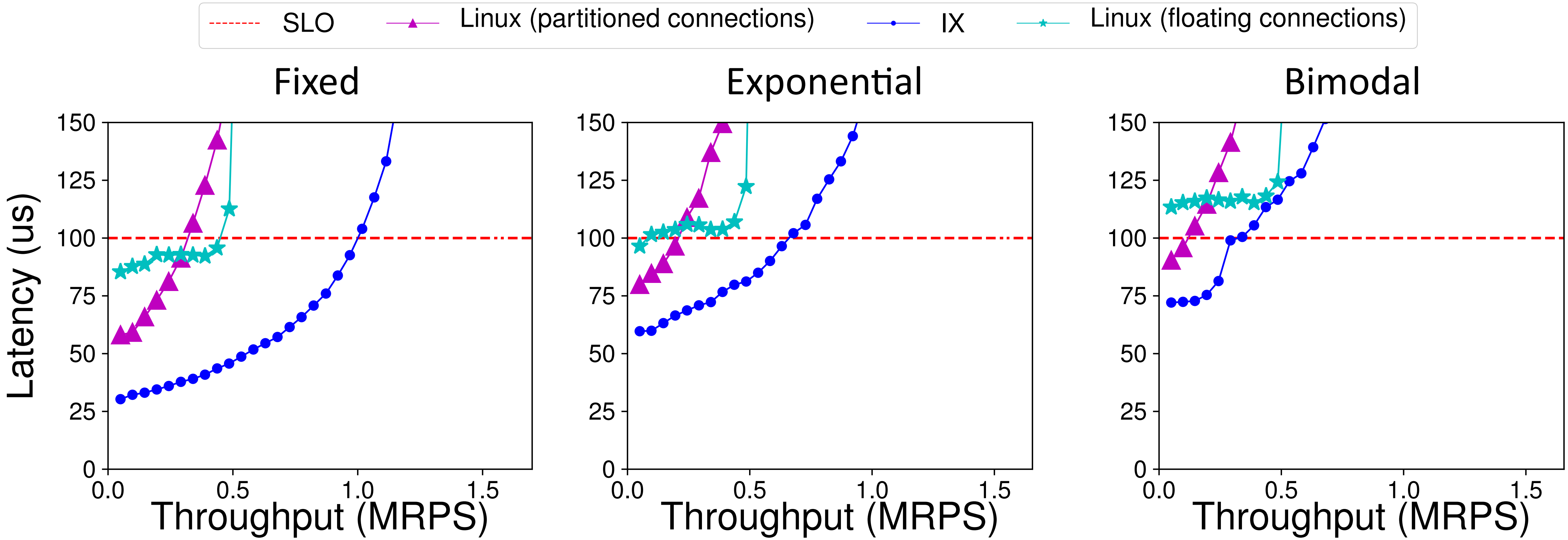
Execution Model



Evaluation Setup

- Environment:
  - 10+1 Xeon Servers
  - 16-hyperthread server machine
  - Quanta/Cumulus 48x10GbE switch
- Experiments:
  - Synthetic micro-benchmarks
  - Silo [SOSP 2013]
  - Memcached
- Baselines:
  - IX
  - Linux (partitioned and floating connections)

Latency vs Load – Service Time 10µs

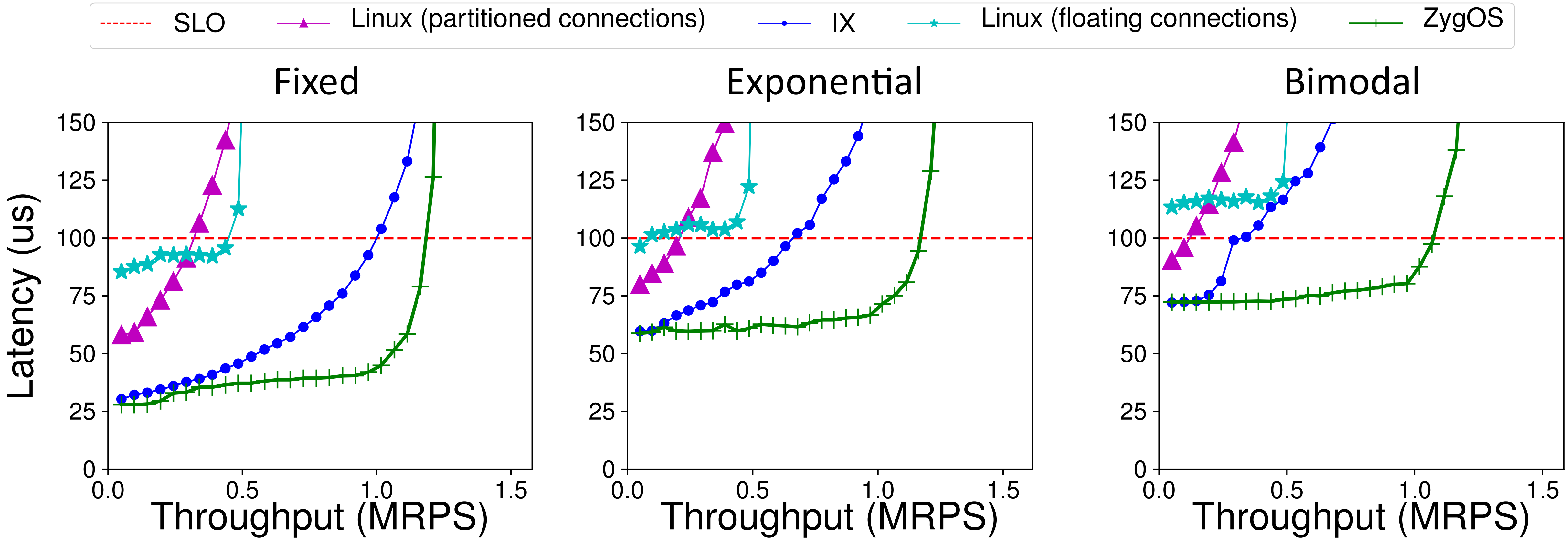


99th percentile latency

SLO: 10 x AVG[service_time]

IX, Belay et al. OSDI 2014

Latency vs Load – Service Time 10µs

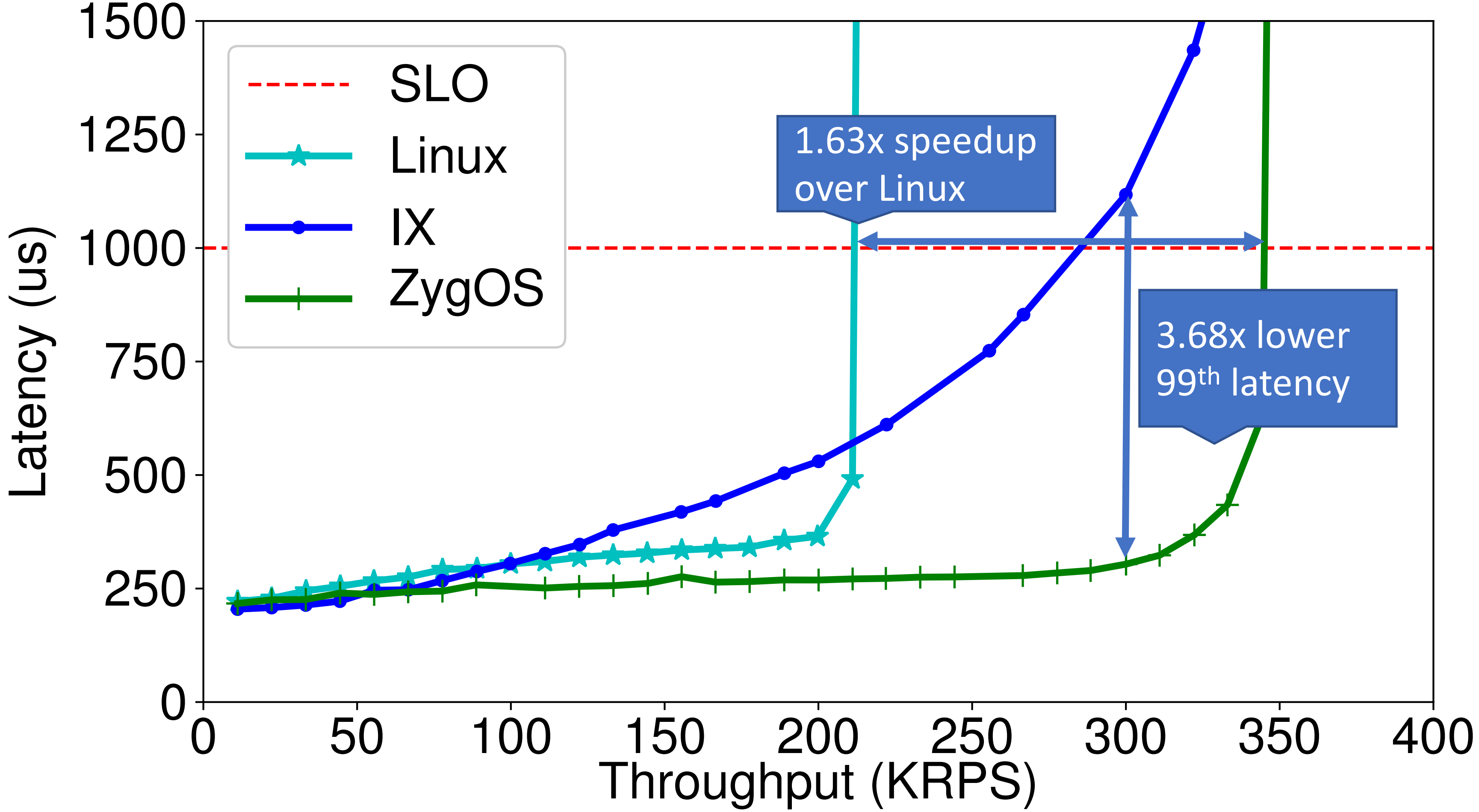


99th percentile latency

SLO: 10 x AVG[service_time]

IX, Belay et al. OSDI 2014

Silo with TPC-C workload



Conclusion

Fork me on GitHub

ZygOS: A datacenter operating system for low-latency

- Reduced System overheads
- Converges to a single queue model
- Work conservation through work stealing
- Reduce HOL through light-weight IPIs

We ♥ opensource

https://github.com/ix-project/zygos

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

Tiresias

A GPU Cluster Manager for Distributed Deep Learning

Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin,
Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang (Harry) Liu, Chuanxiong Guo

UNIVERSITY OF MICHIGAN

Microsoft ByteDance UNIST ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY Alibaba Group

GPU Cluster for Deep Learning Training

- Deep learning (DL) is popular
  - *10.5×* increase of DL training jobs in Microsoft
  - DL training jobs require GPU
    - Distributed deep learning (DDL) training with multiple GPUs

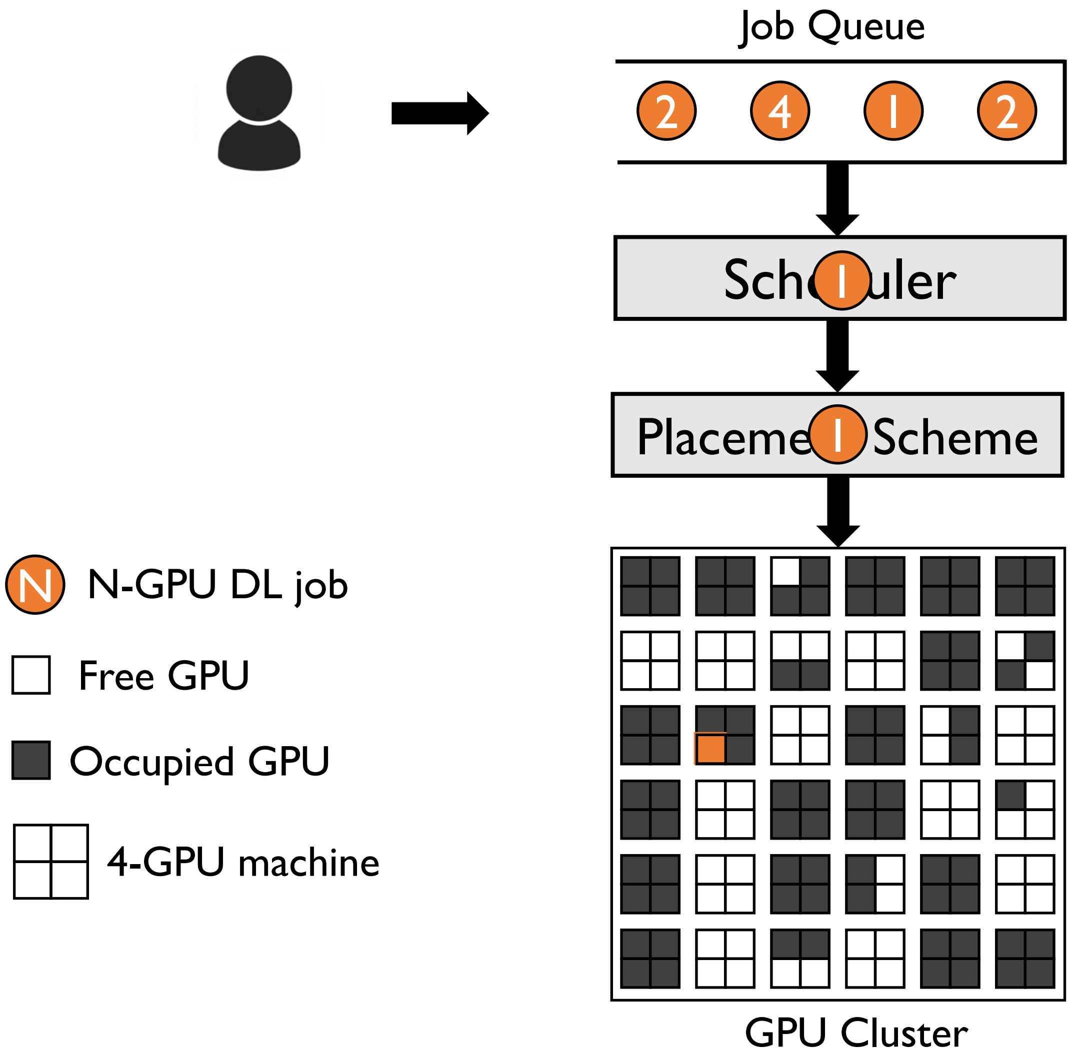
Google Lens

Siri

- GPU cluster for DL training
  - *5×* increase of GPU cluster scale in Microsoft [1]

How to efficiently manage a GPU cluster for DL training jobs?

GPU Cluster Manager



Design Objectives

Minimize

Cluster-Wide Average Job Completion Time (JCT)

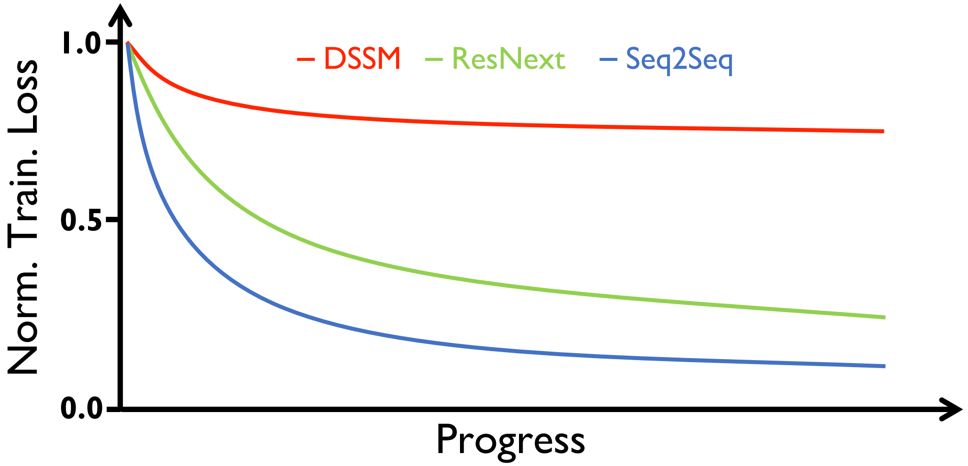
Achieve

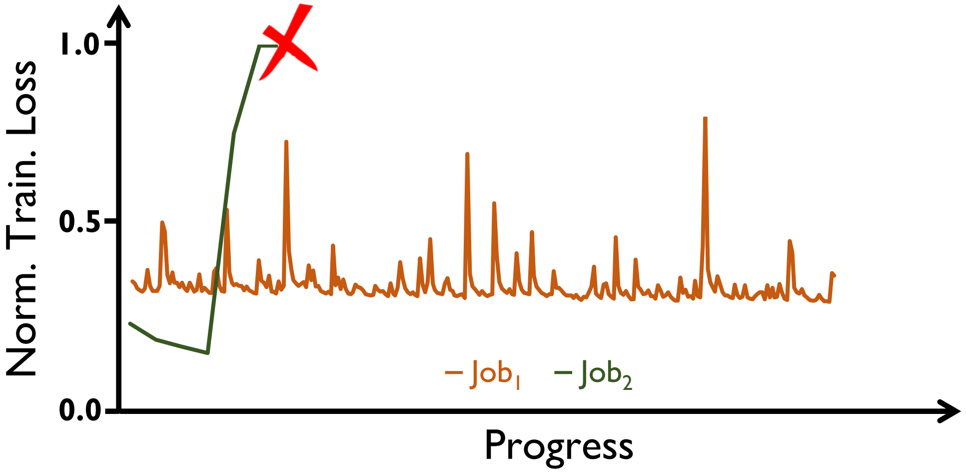
High Resource (GPU) Utilization

Challenge 1: Unpredictable Training Time

- Unknown execution time of DL training jobs
  - Job execution time is useful when minimizing JCT

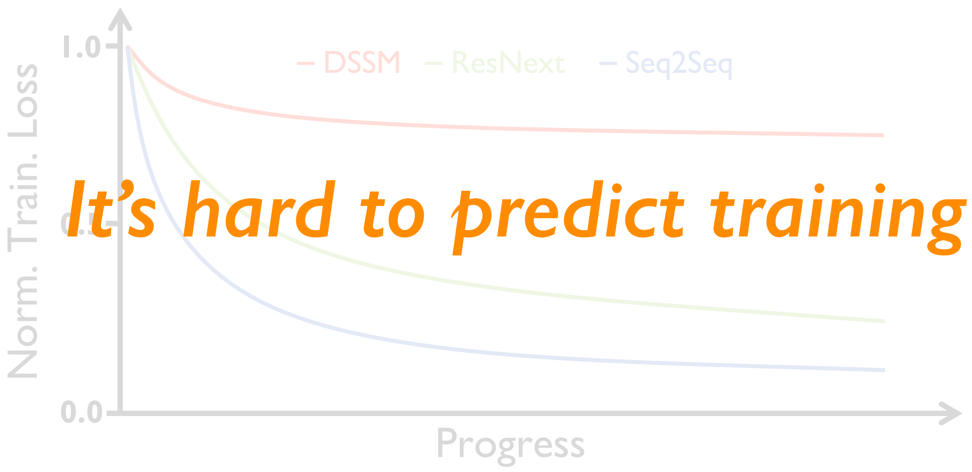
- Predict job execution time
  - Use the smooth loss curve of DL training jobs (*Optimus* [1])





Challenge 1: Unpredictable Training Time

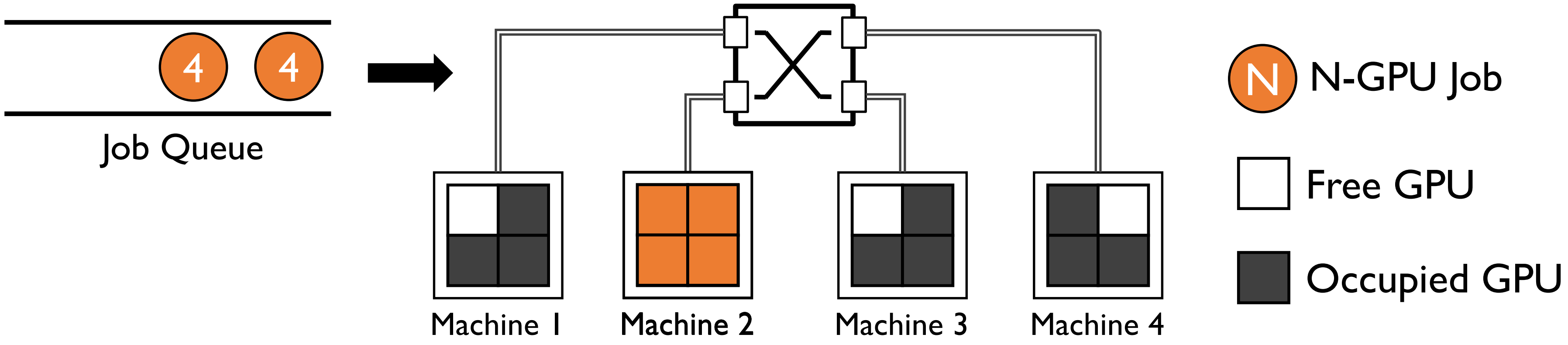
- Unknown execution time of DL training jobs
  - Job execution time is useful when minimizing JCT
- Predict job execution time
  - Use the smooth loss curve of DL training jobs (*Optimus* [1])



It's hard to predict training time of DL jobs in many cases

Challenge II: Over-Aggressive Job Consolidation

- Network overhead in DDL training
- *Consolidated placement* for good training performance
  - *Fragmented free GPUs in the cluster*
  - *Longer queuing delay*



Prior Solutions

	I. *Unpredictable Training Time* (*Scheduling*)	II. *Over-Aggressive Job Consolidation* (*Job Placement*)
Optimus[1]	None	None
YARN-CS	*FIFO*	None
Gandiva[2]	*Time-sharing*	*Trial-and-error*

[1]. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters, EuroSys'18
[2]. Gandiva: Introspective Cluster Scheduling for Deep Learning, OSDI'18

Tiresias

A GPU cluster manager for Distributed Deep Learning Without Complete Knowledge

1. Age-Based Scheduler

   Minimize JCT without complete knowledge of jobs

2. Model Profile-Based Placement

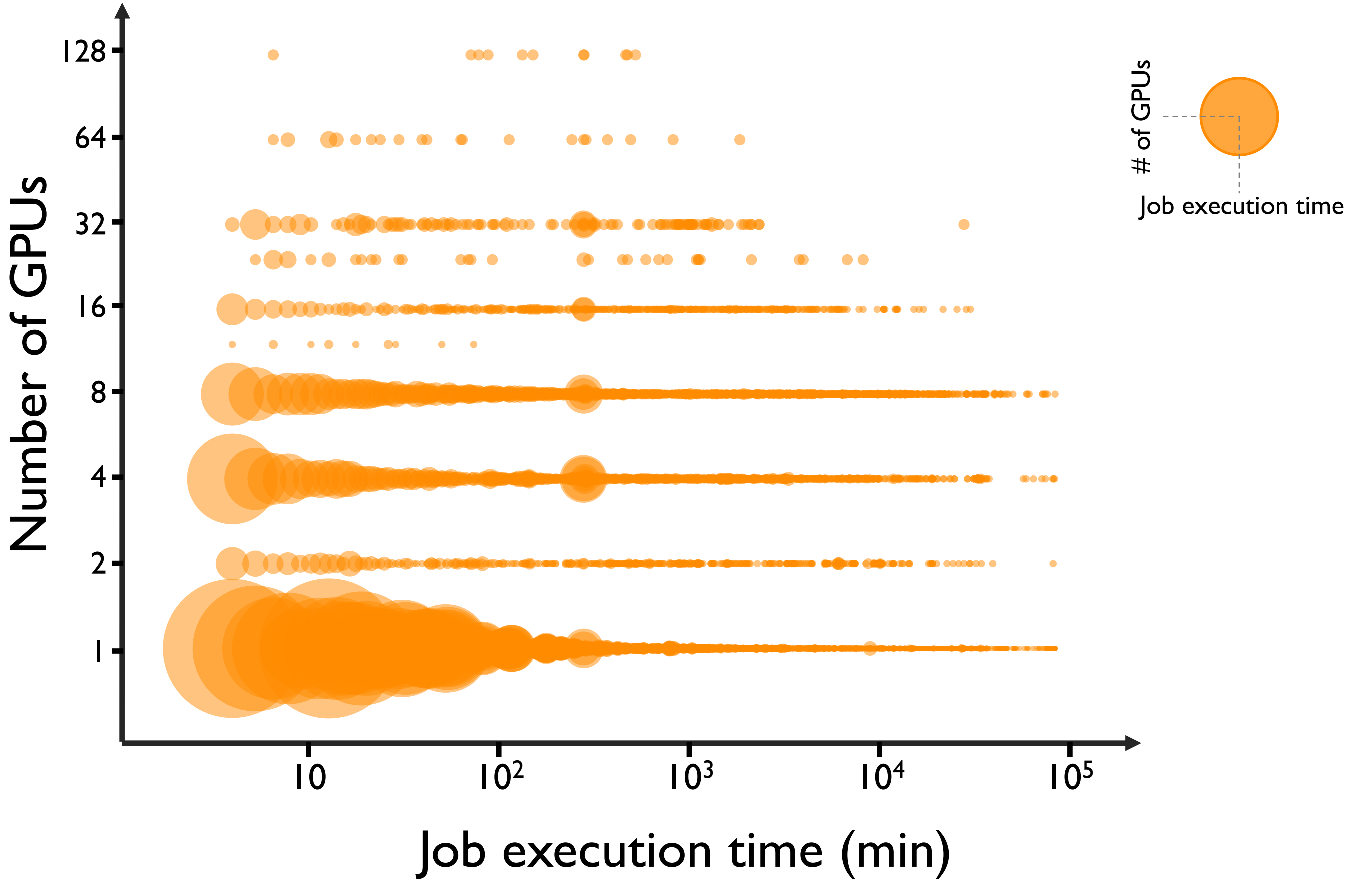
   Place jobs without additional information from users

Challenge 1

How To Schedule DL Training Jobs Without Complete Job Information?

Characteristics of DL Training Jobs

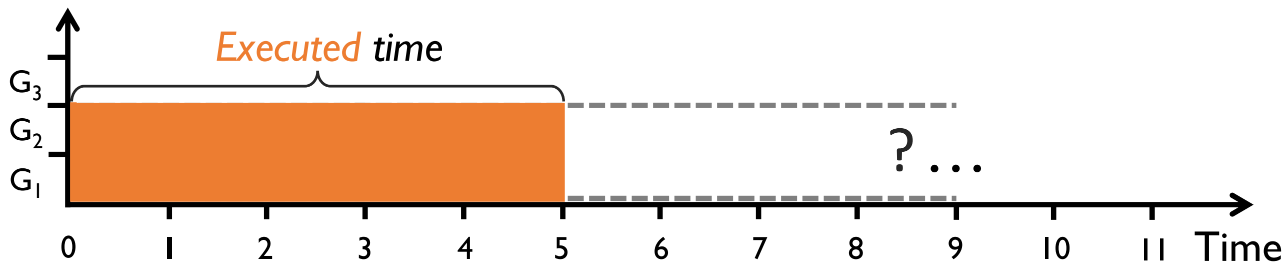
- Variations in both temporal and spatial aspects



Scheduler should consider both ***temporal and spatial*** *aspects of DL training jobs*

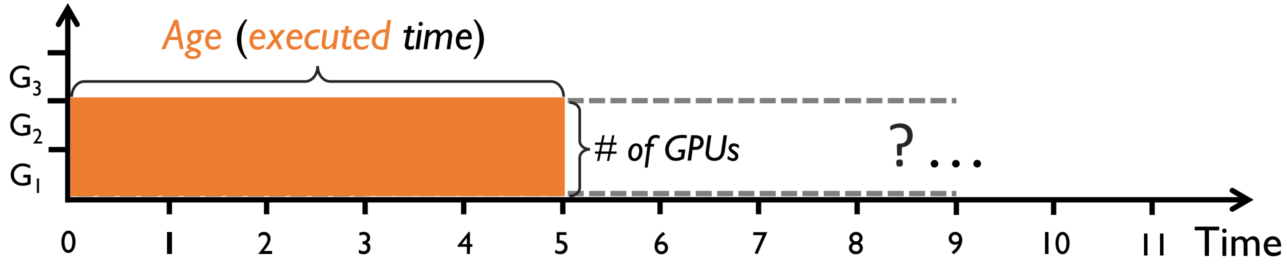
Available Job Information

1. Spatial: number of GPUs
2. Temporal: *executed* time



Age-Based Schedulers

- ***Least-Attained Service***[1] (LAS)
  - Prioritize job that has the shortest executed time



[1]. Feedback queueing models for time-shared systems. JACM, 1968
[2]. Multi-armed bandit allocation indices. Wiley, Chichester, 1989

Two-Dimensional Age-Based Scheduler (2DAS)

- Age calculated by two-dimensional attained service
  - i.e., a job's ***total executed GPU time*** (# of GPUs × executed time)

- No prior information
  - *2D-LAS*

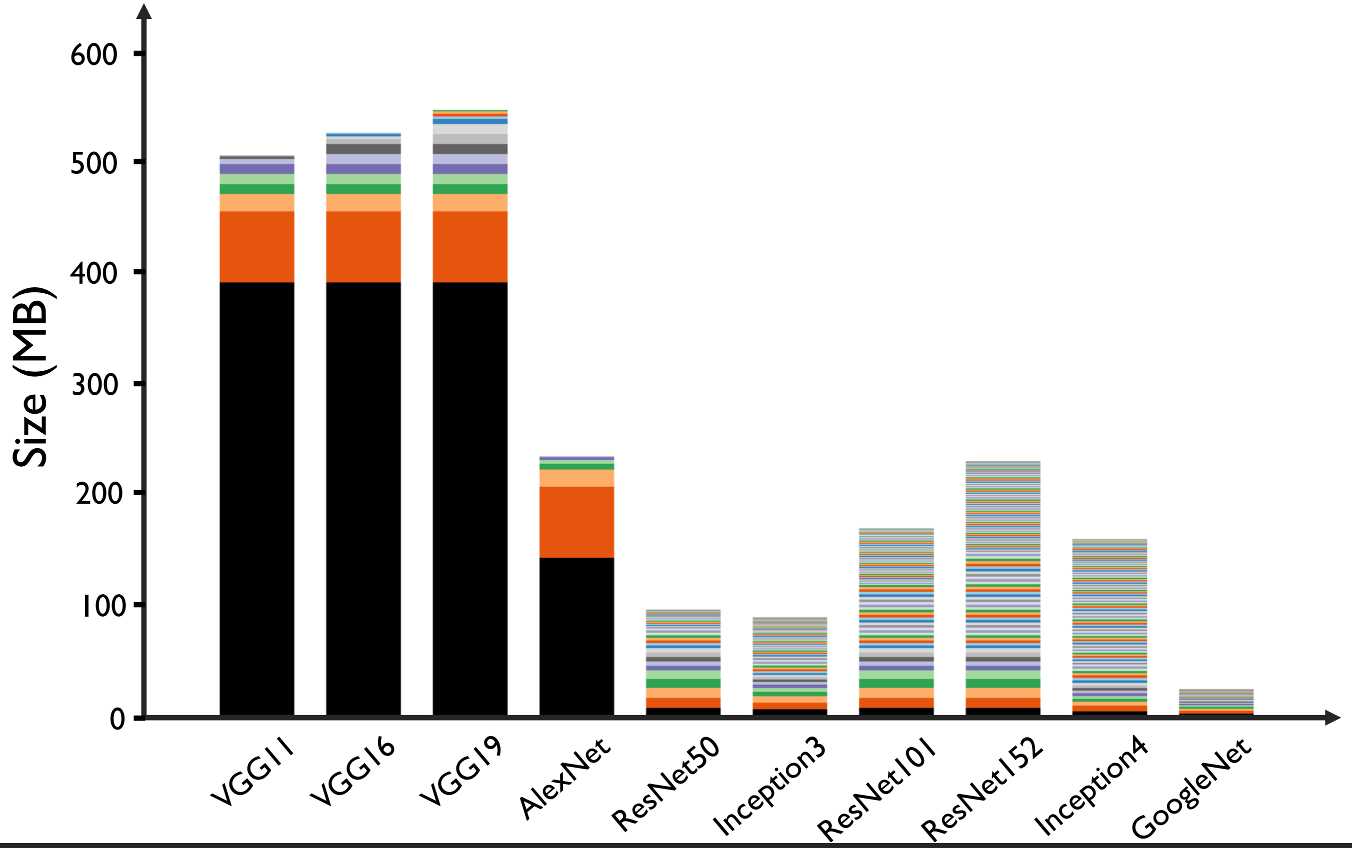
Fewer Job Switches: Discretized 2D-LAS (MLFQ)

Challenge II

How to Place DL Jobs Without Hurting Training Performance?

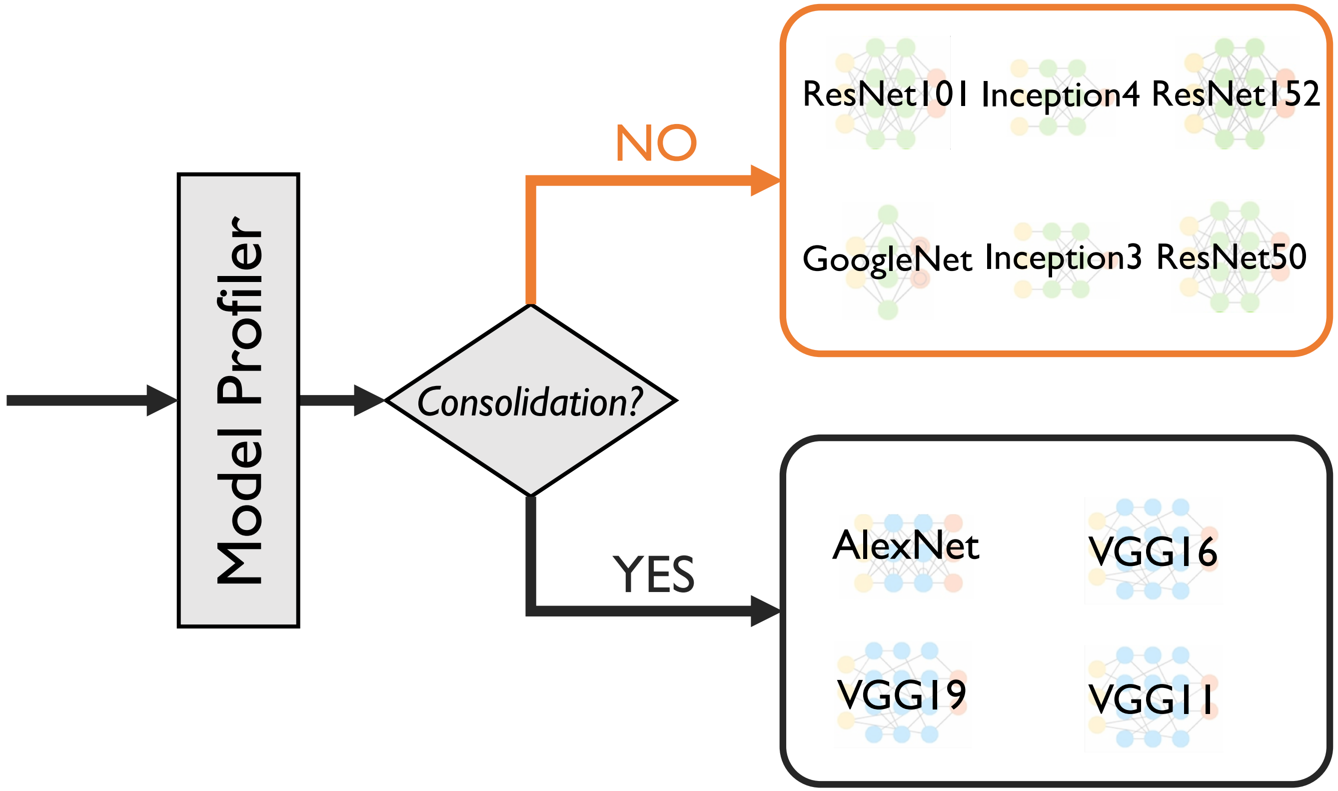
Characteristics of DL Models

- Tensor size in DL models
  - *Large tensors* cause network imbalance and contention



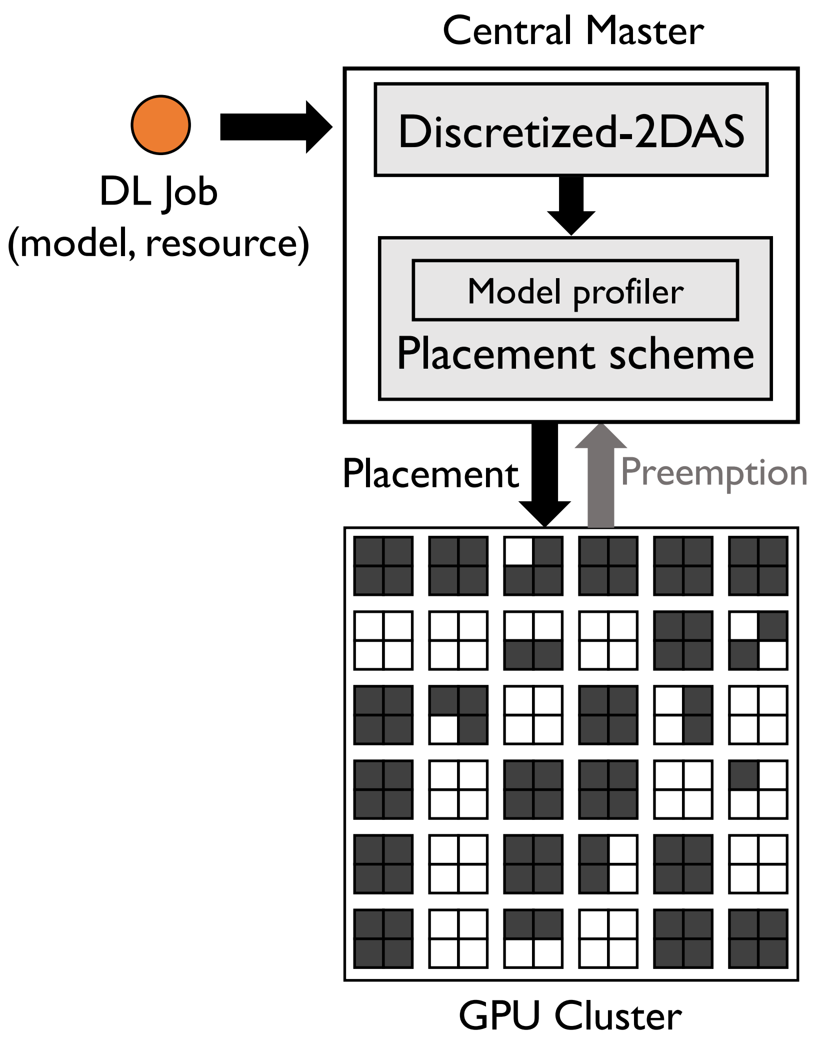
Consolidated placement is needed when the model is highly skewed in its tensor size

Model Profile-Based Placement



Tiresias

Central Master
Network-Level Model Profiler



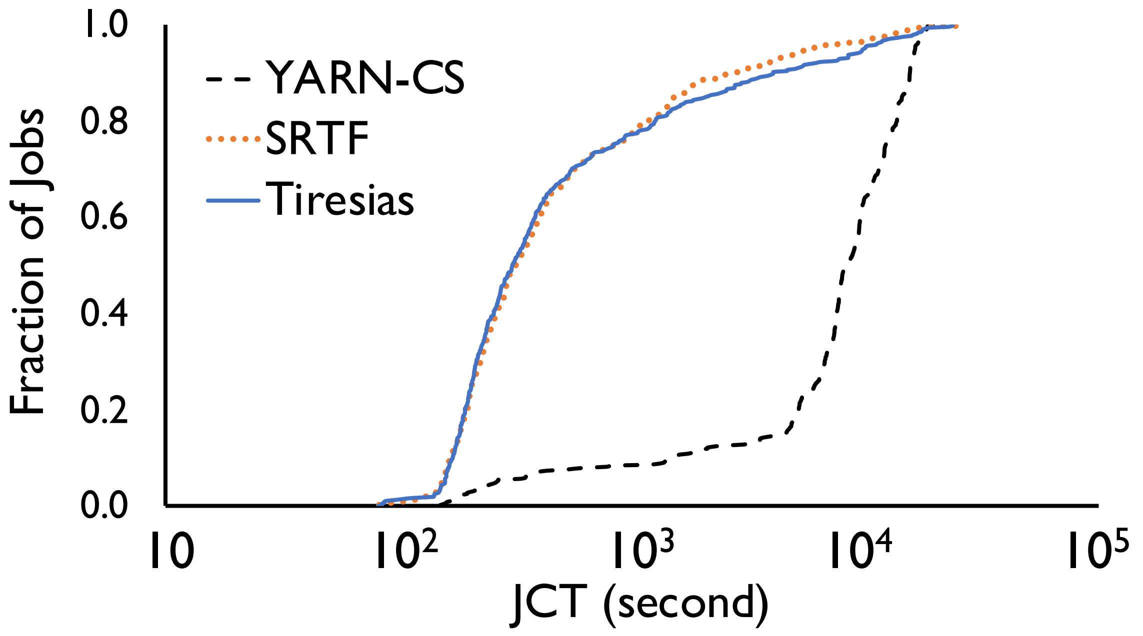
Evaluation

60-GPU
Testbed Experiment

Large-scale &
Trace-driven Simulation

JCT Improvements in Testbed Experiment

- Testbed – Michigan ConFlux cluster
  - 15 machines (4 GPUs each)
  - 100 Gbps RDMA network

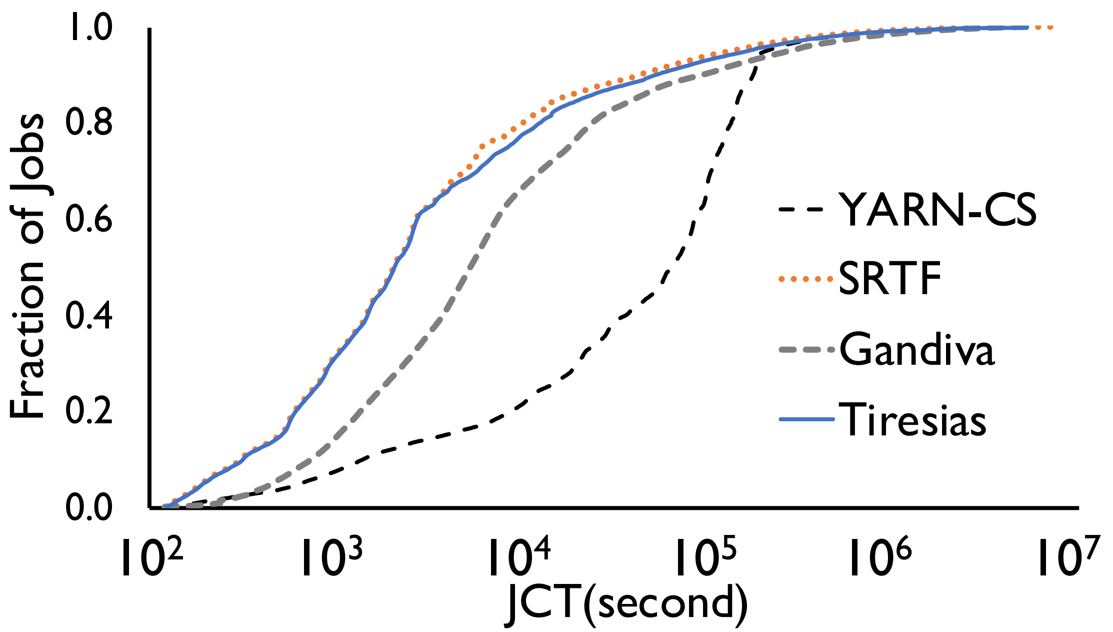


Avg. JCT improvement (w.r.t. YARN-CS): 5.5×

Comparable performance to SRTF

JCT Improvements in Trace-Driven Simulation

- Discrete-time simulator
  - 10-week job trace from Microsoft
  - 2,000-GPU cluster



Avg. JCT improvement (w.r.t. Gandiva): 2×

Tiresias

A GPU cluster manager for Distributed Deep Learning Without Complete Knowledge

- Optimize JCT with no or partial job information
- Relax placement constraint without hurting training performance
- Simple, practical, and with significant performance improvements

UNIVERSITY OF MICHIGAN

https://github.com/SymbioticLab/Tiresias

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

Dominant Resource Fairness (DRF)

Fair Allocation of Multiple Resource Types

Ali Ghodsi, Matei Zaharia
Benjamin Hindman, Andy Konwinski,
Scott Shenker, Ion Stoica

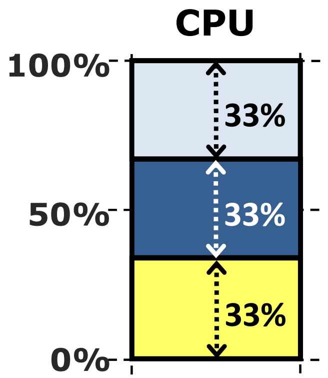
University of California, Berkeley

alig@cs.berkeley.edu

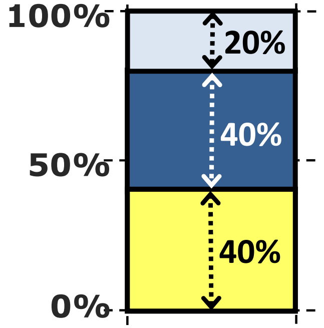
What is fair sharing?

- n users want to share a resource (e.g. CPU)
  - Solution:

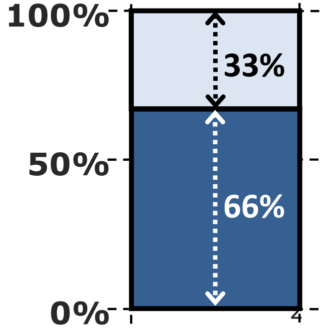
    Allocate each 1/n of the shared resource



- *Generalized by* *max-min fairness*
  - Handles if a user wants less than its fair share
  - E.g. user 1 wants no more than 20%



- Generalized by *weighted max-min fairness*
  - Give weights to users according to importance
  - User 1 gets weight 1, user 2 weight 2



alig@cs.berkeley.edu

How to define fairness?

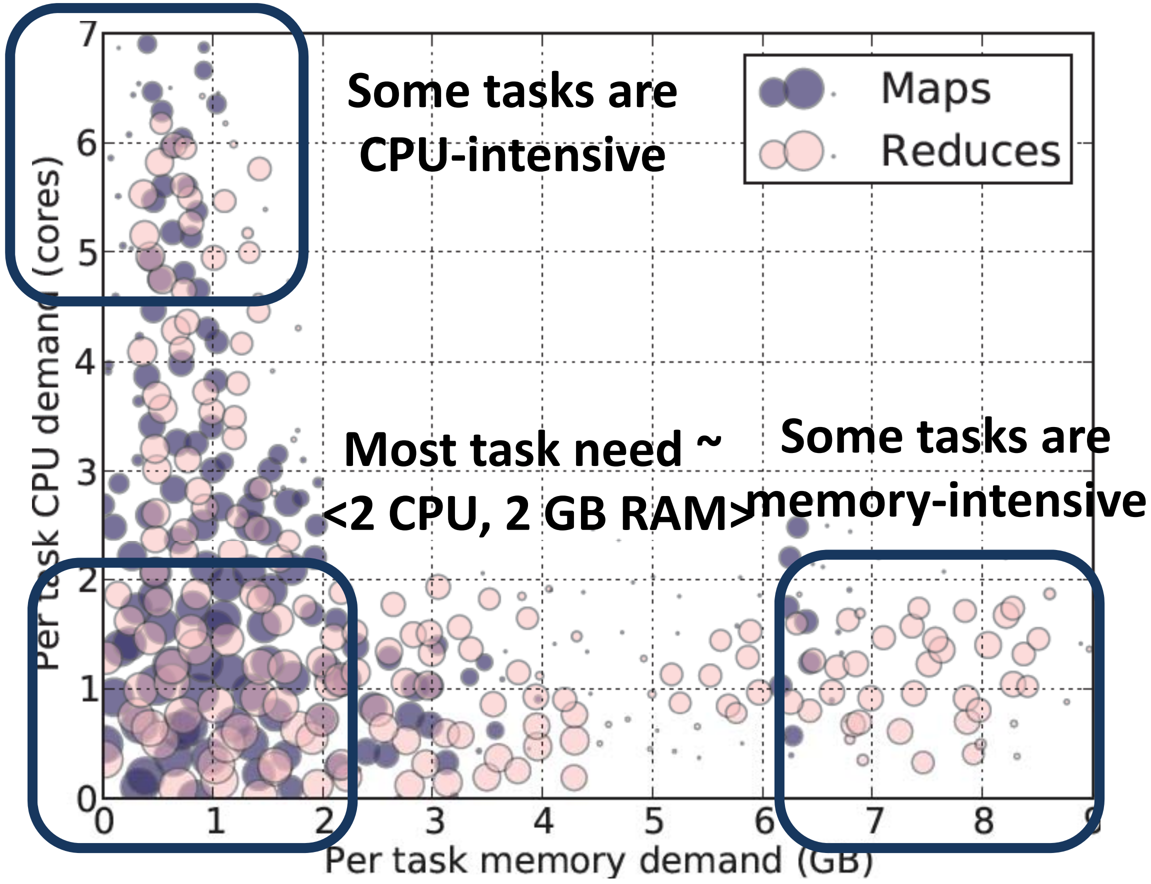
- Share guarantee
  - Each user can get at least 1/n of the resource
  - But will get less if her demand is less
- Stragegy-proof
  - Users are not better off by asking for more than they need
  - Users have no reason to lie
- Pareto efficiency
  - It is not possible to increase the allocation of a user without decreasing the allocation of at least another user
  - It leads to maximizing system utilizaiton subject to satisfying other constraints

Why is max-min fairness not enough?

- Job scheduling in datacenters is not only about CPUs
  - Jobs consume CPU, memory, disk, and I/O

- Does this pose any challenge?

Heterogeneous Resource Demands



2000-node Hadoop Cluster at Facebook (Oct 2010)

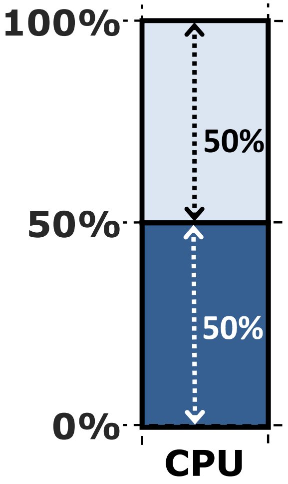
Problem

Single resource example

– 1 resource: CPU

– User 1 wants **<1 CPU>** per task

– User 2 wants **<3 CPU>** per task



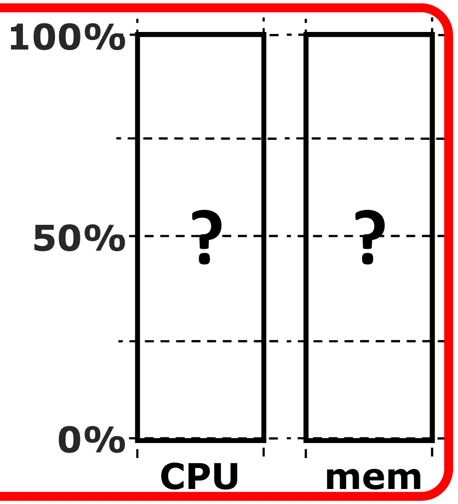
Multi-resource example

– 2 resources: CPUs & mem

– User 1 wants **<1 CPU, 4 GB>** per task

– User 2 wants **<3 CPU, 1 GB>** per task

– ***What's a fair allocation?***



Problem definition

How to fairly share multiple resources when users have heterogenous demands on them?

Model

- Users have *tasks* according to a *demand vector*
  - e.g. **<2, 3, 1>** user's tasks need 2 R_1, 3 R_2, 1 R_3
  - Not needed in practice, measure actual consumption
- Resources given in multiples of demand vectors
- Assume divisible resources

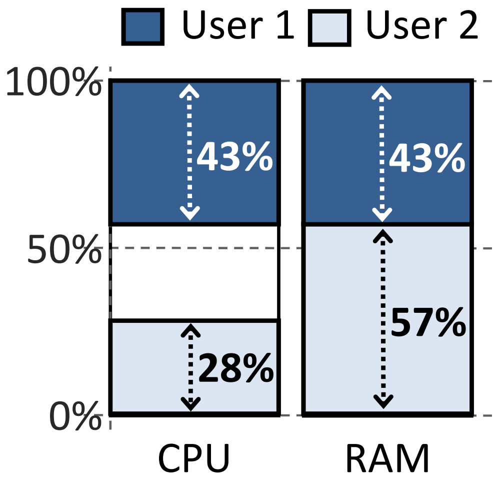
A Natural Policy

- *Asset Fairness*
  - Equalize each user's *sum of resource shares*

- Cluster with 70 CPUs, 70 GB RAM
  - U_1 needs <2 CPU, 2 GB RAM> per task
  - U_2 needs <1 CPU, 2 GB RAM> per task

A Natural Policy

- *Asset Fairness*
  - Equalize each user's *sum of resource shares*



Problem

User 1 has < 50% of both CPUs and RAM

Better off in a separate cluster with 50% of the resources

- Asset fairness yields
  - U_1: 15 tasks: 30 CPUs, 30 GB (∑=60)
  - U_2: 20 tasks: 20 CPUs, 40 GB (∑=60)

Dominant Resource Fairness

- A user's *dominant resource* is the resource she has the biggest share of
  - Example:

    Total resources: **<10 CPU, 4 GB>**

    User 1's allocation: **<2 CPU, 1 GB>**

    Dominant resource is memory as 1/4 > 2/10 (1/5)

- A user's *dominant share* is the fraction of the dominant resource she is allocated
  - User 1's dominant share is **25%** (1/4)

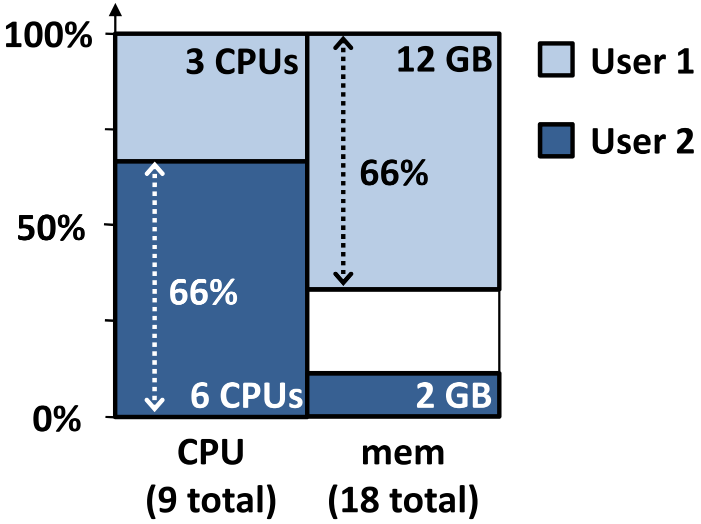
Dominant Resource Fairness (2)

- ***Apply max-min fairness to dominant shares***
- Equalize the dominant share of the users
  - Example:

    Total resources: **<9 CPU, 18 GB>**

    User 1 demand: **<1 CPU, 4 GB>** dom res: **mem**

    User 2 demand: **<3 CPU, 1 GB>** dom res: **CPU**



Properties of Policies

Property	Asset	CEEI	DRF
Share guarantee		✔	✔
Strategy-proofness	✔		✔
Pareto efficiency	✔	✔	✔
Envy-freeness	✔	✔	✔
Single resource fairness	✔	✔	✔
Bottleneck res. fairness		✔	✔
Population monotonicity	✔		✔
Resource monotonicity			

Scheduling in Modern Computer Systems

- FCFS
  - SOSP’17 ZygOS
- RR
  - NSDI’19 Shinjuku
- MLFQ
  - NSDI’19 Tiresias
- Fairness
  - NSDI’11 DRF
  - NSDI’16 FairRide

UC BERKELEY

FairRide: Near-Optimal Fair Cache Sharing

Qifan Pu,
Haoyuan Li,
Matei Zaharia,
Ali Ghodsi,
Ion Stoica

Caches are crucial

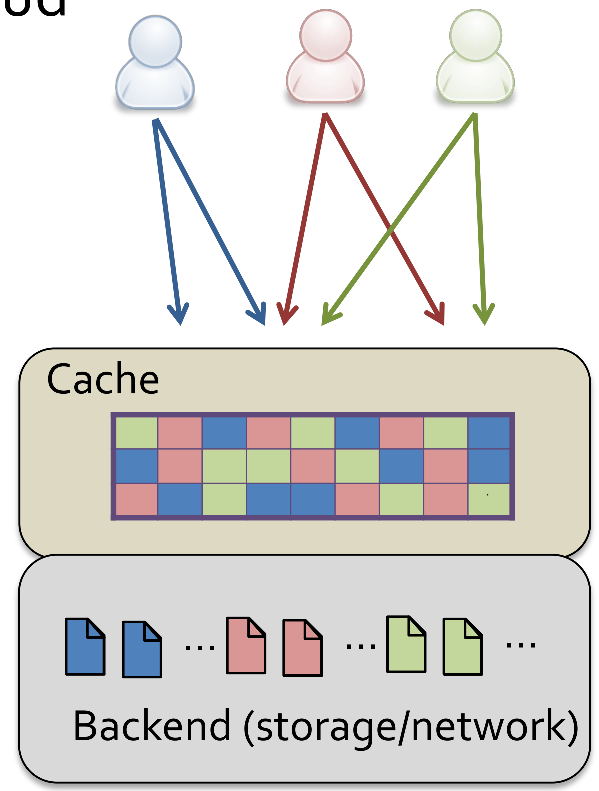
ALLUXIO

Spark

memCached

Cache sharing

- Increasingly, caches are shared among multiple users
  – Especially with the advent of cloud

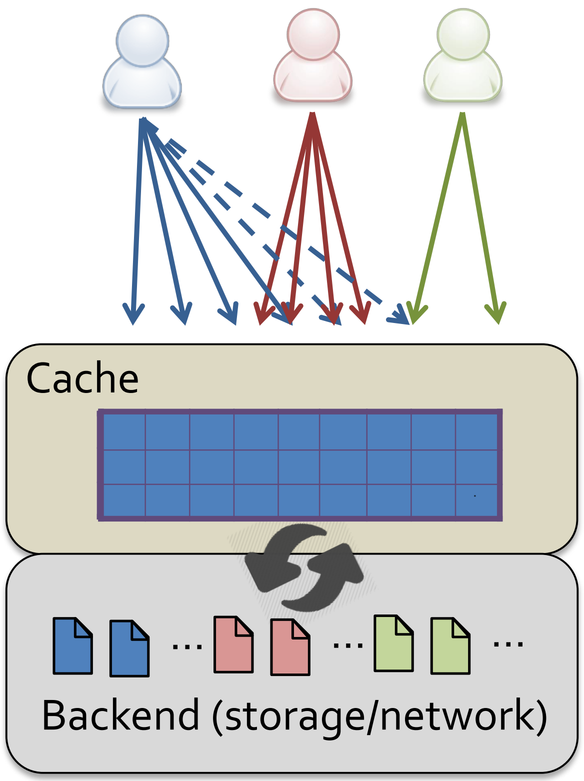


Benefits:

– Provide low latency

– Reduce backend load

Problems with cache algorithms



- LRU, LFU, LRU-K…
  - Cache data likely to be accessed in the future
- Optimize global efficiency
- Single user gets arbitrarily small cache
- Prone to strategic behavior

A simple model

- Users access equal-sized files at constant rates
  - r_{ij} the rate user *i* accesses file *j*

- A allocation **policy** decides which files to cache
  - p_j the % of file *j* put in cache

- Users care their hit ratio
  - user *i*'s hit ratio: $HR_i = \frac{total\_hits}{total\_accesses} = \frac{\sum_j p_j r_{ij}}{\sum_j r_{ij}}$

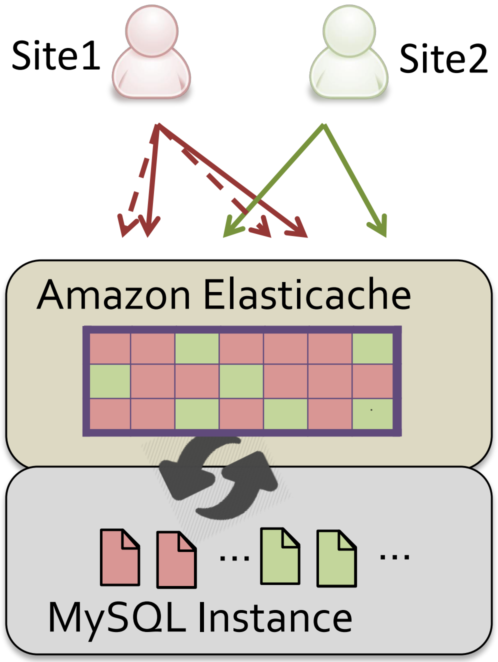
◆ Results hold with varied file sizes, access partial files, p_j is binary, etc.

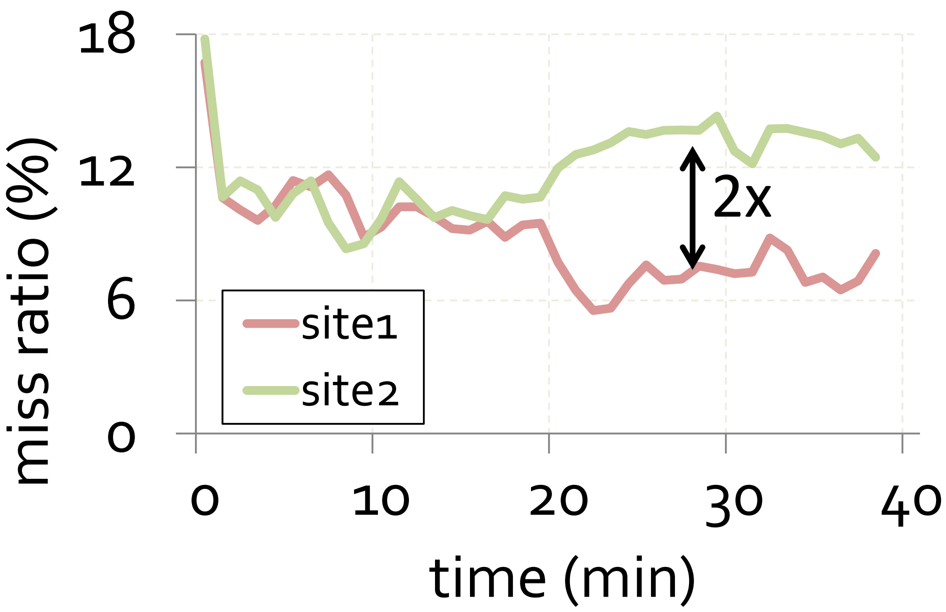
Properties

- Isolation Guarantee **(Share Guarantee)**
  – No user should be worse off than static allocation
- Strategy-Proofness
  – No user can improve by cheating
- Pareto Efficiency
  – Can't improve a user without hurting others

Strategy proofness

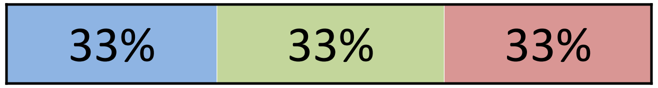
- Very easy to cheat, hard to detect
  - e.g., by making spurious accesses
- Can happen in practice



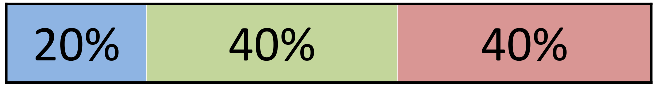


What is *max-min fairness?*

- *Maximize* the the user with *minimum* allocation
  - Solution: allocate each **1/n** (fair share)

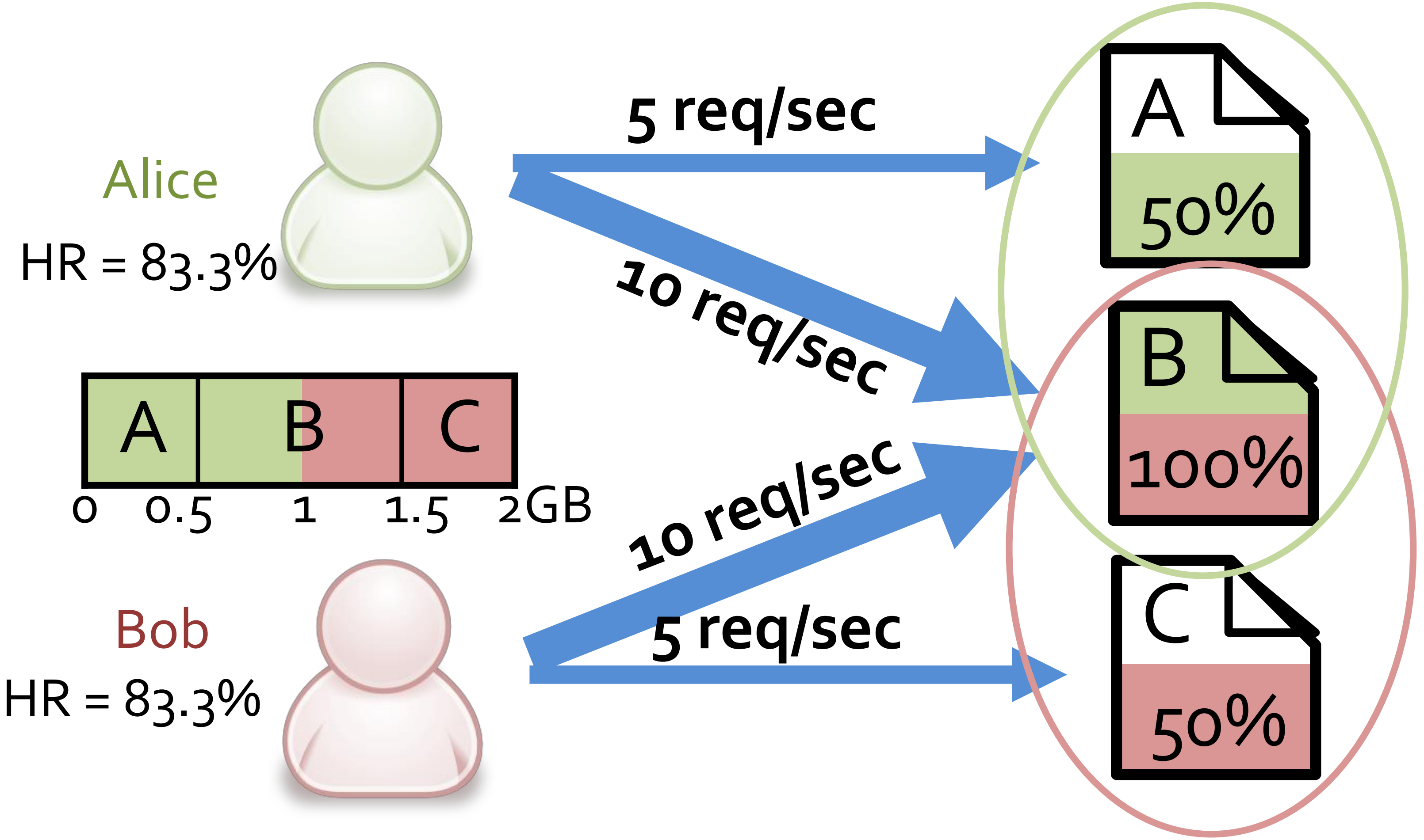


  - Handles if some users want less than fair share



- Widely successful to other resources:
  - OS: round robin, prop sharing, lottery sched…
  - Networking: fair queueing, wfq, wf2q, csfq, drr…
  - Datacenter: DRF, Hadoop fair sched, Quincy…

An example

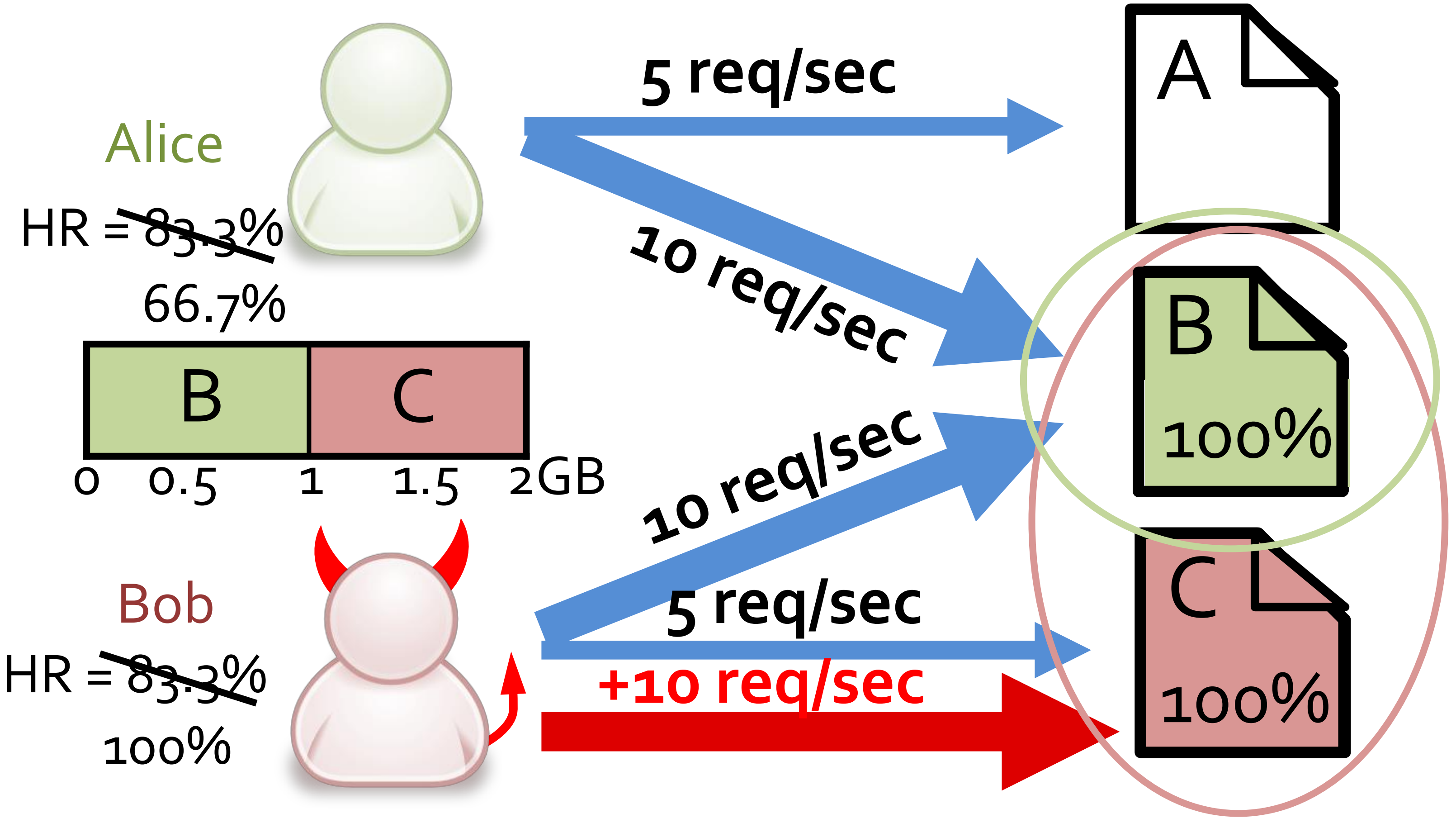


file sizes = 1GB, total cache = 2GB

Properties

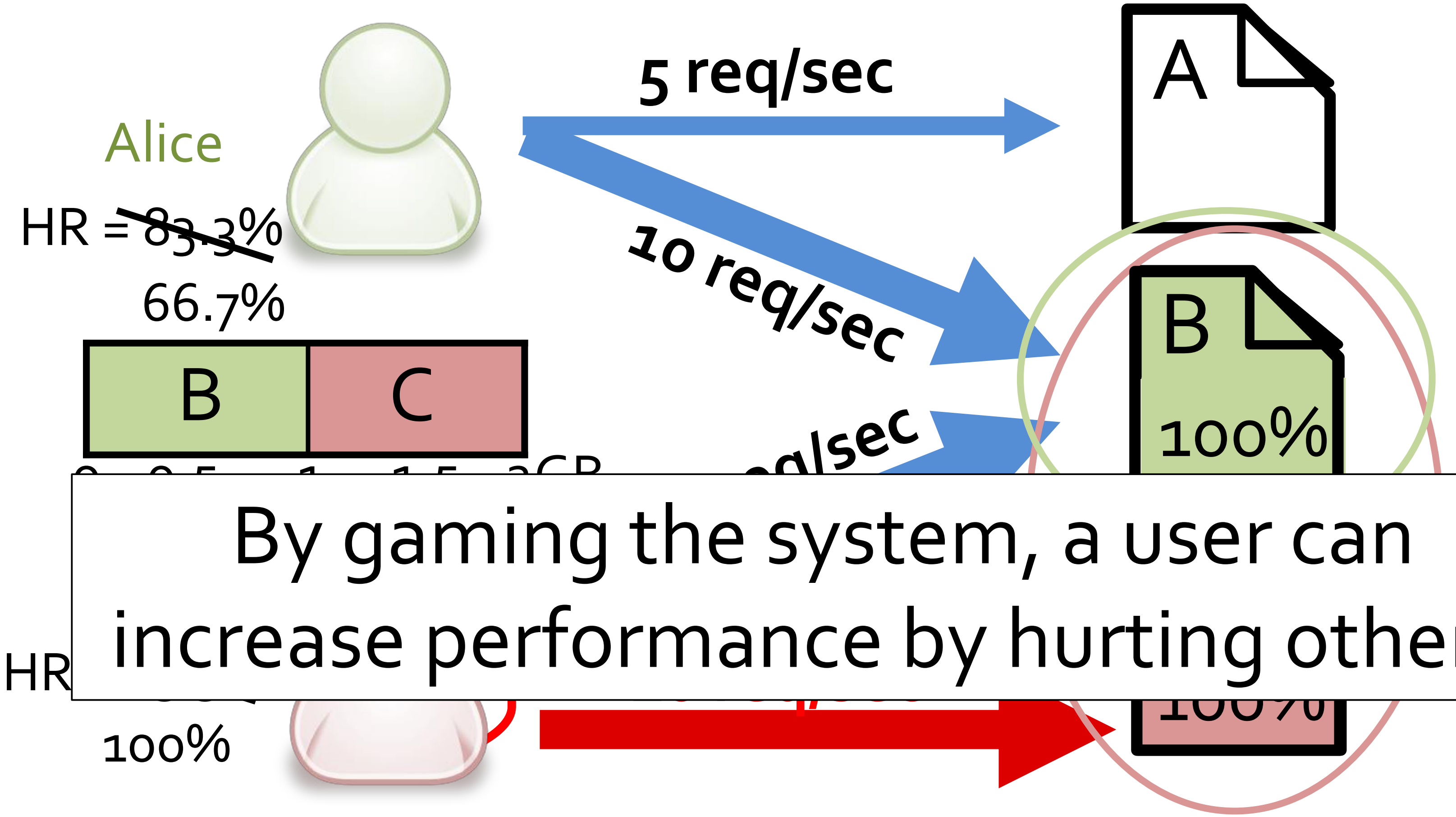
	Isolation Guarantee	Strategy Proofness	Pareto Efficiency
max-min fairness	✓	?	✓

An example



file sizes = 1GB, total cache = 2GB

An example



By gaming the system, a user can increase performance by hurting others!

file sizes = 1GB, total cache = 2GB

Properties

	Isolation Guarantee	Strategy Proofness	Pareto Efficiency
max-min fairness	✓	✗	✓
static allocation	✓	✓	✗
priority allocation	✗	✓	✓
max-min rate	✗	✓	✗
…	…	…	…

Theorem

No allocation policy can satisfy **all three** properties!

- Best we can do: two of three.

FairRide

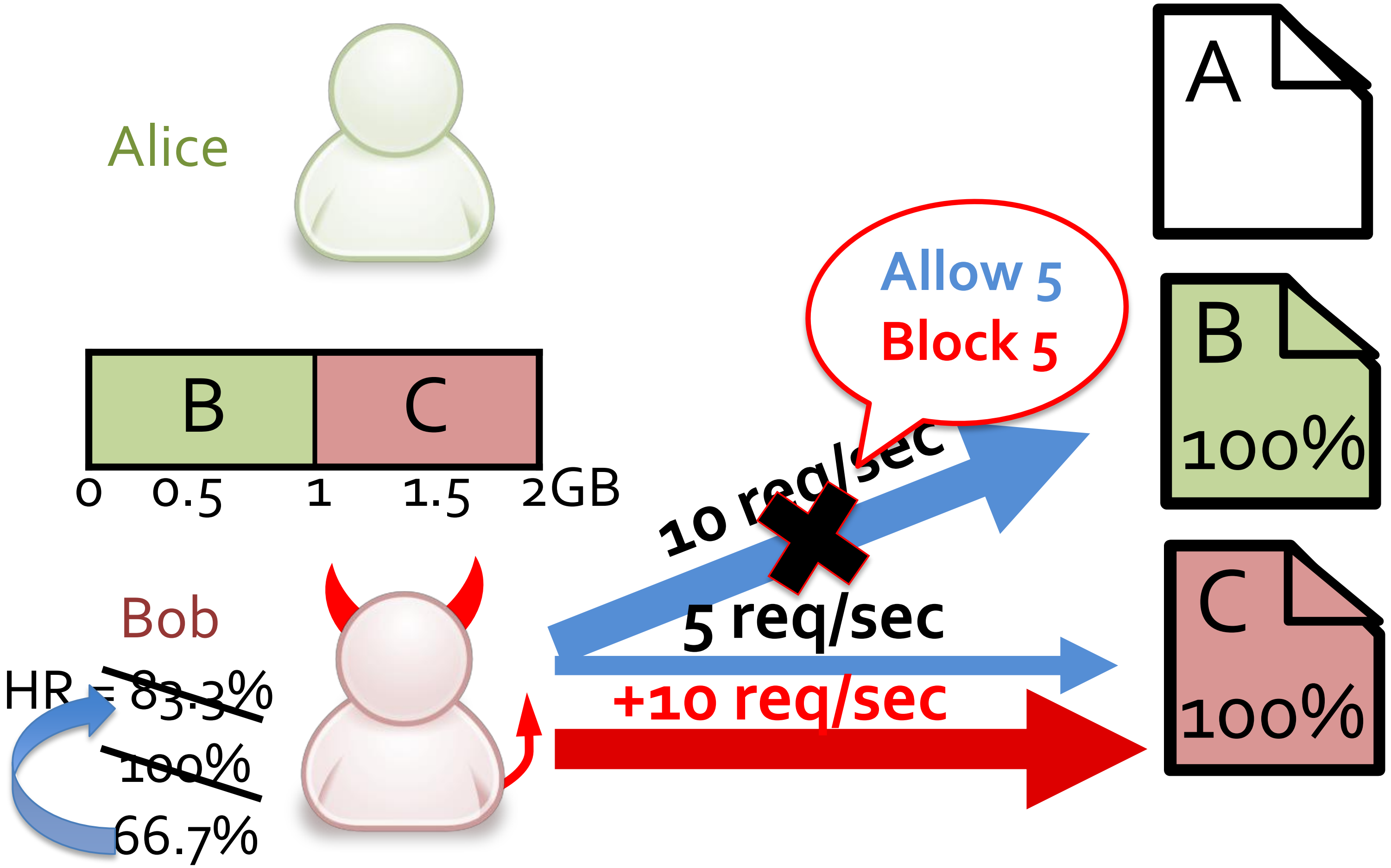
- Starts with max-min fairness
  - Allocate 1/n to each user
  - Split "cost" of shared files equally among shared users

- Only difference:

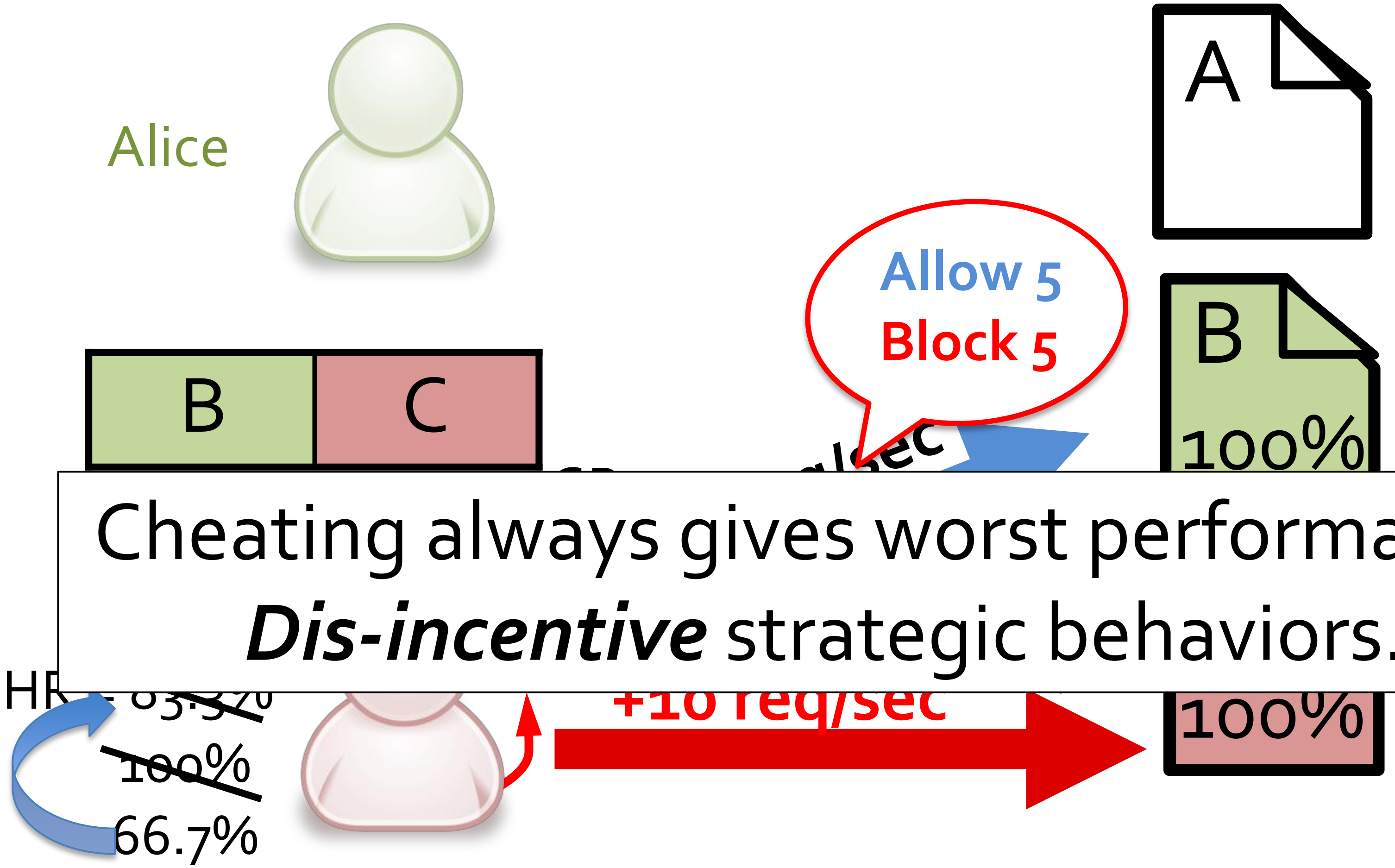
  blocking users who don't "pay" from accessing

- Probabilistic blocking: with some probability
  - Implemented with delaying

FairRide: Blocking



FairRide: Blocking



Cheating always gives worst performance.

Dis-incentive strategic behaviors.

Probabilistic blocking

- FairRide blocks a user with ***p(nj) = 1/(nj+1)*** probability
  - *nj* is number of other users caching file *j*
  - e.g., p(1)=50%, p(4)=20%

- The best you can do in a general case
  - **Less blocking does not prevent cheating**

Properties

	Isolation Guarantee	Strategy Proofness	Pareto Efficiency
max-min fairness	✓	✗	✓
static allocation	✓	✓	✗
priority allocation	✗	✓	✓
max-min rate	✗	✓	✗
FairRide	✓	✓	Near-optimal