# CE 528 Cloud Computing

Lecture 15: Performance Issue
Spring 2026

**Prof. Yigong Hu**

Slides courtesy of Amanda Raybuck and Key Ousterhout

# Latency Numbers

| L1 Cache Reference | 0.5 ns | | | |
|---|---|---|---|---|
| Branch Mispredict | 5 ns | | | |
| L2 Cache Reference | 7 ns | | | 14x L1 cache |
| Mutex Lock/Unlock | 25 ns | | | |
| Main Memory Reference | 100 ns | | | 20x L2 cache, 200x L1 cache |
| Compress 1K Bytes with Zippy | 3,000 ns | 3 μs | | |
| Send 1K Bytes over 1 Gbps Network | 10,000 ns | 10 μs | | |
| Read 4K Randomly from an SSD | 150,000 ns | 150 μs | | ~1GB/s SSD |
| Read 1MB Sequentially from Memory | 250,000 ns | 250 μs | | |
| Round Trip within Same Datacenter | 500,000 ns | 500 μs | | |
| Read 1MB Sequentially from SSD | 1,000,000 ns | 1,000 μs | 1 ms | ~1GB/s SSD, 4X memory |
| Disk Seek | 10,000,000 ns | 10,000 μs | 10 ms | 20x datacenter roundtrip |
| Read 1MB Sequentially from Disk | 20,000,000 ns | 20,000 μs | 20 ms | 80x memory, 20x SSD |
| Send Packet CA->Netherlands->CA | 150,000,000 ns | 150,000 μs | 150 ms | |

# Handling millisecond-scale request

## Millisecond-scale events

- Disk reads – 10s of ms
- Wide-area network traffic – 10s of ms
- Low-end flash – a few ms

## Software can efficiently mask these

- OS can context switch to a different thread (microseconds)

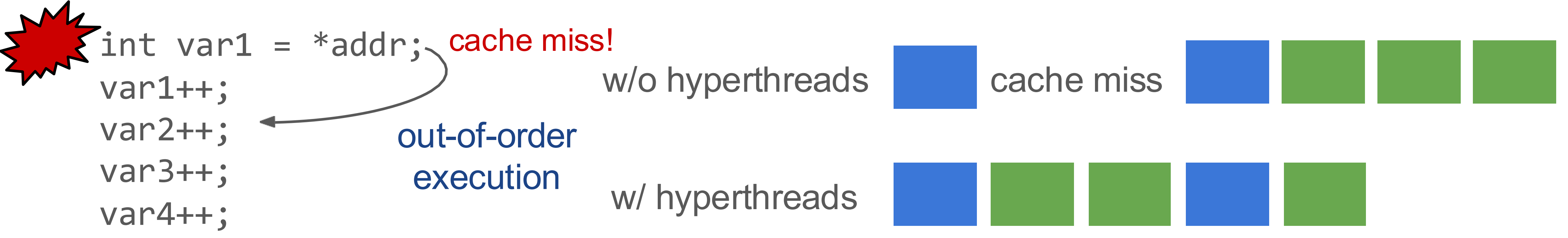
## Programmers can use convenient synchronous (blocking) programming models

# Hardware for millisecond-scale Request

**Hardware can efficiently handle nanosecond-scale events (e.g., cache misses, ~100 ns)**

- Out of order execution,
- Hyperthreads, Simultaneous Multithreading (SMT)
- Prefetching

**Programmers don't have to think about this**



# The Challenge of microsecond-scale Events

**But microsecond-scale events is challenging**

**Datacenter RTT - a few μs**

- High-end flash – tens of μs
- GPU/accelerator - tens of μs

**Hardware techniques do not scale well**

- Not enough independent instructions to fill pipelines for μs
- Not enough hyperthreads to hide μs

**Software techniques have too high of overhead**

- Context switch time dominates microsecond-scale events
- These are the killer microseconds!

# Can Asynchronous Programming Models Help?

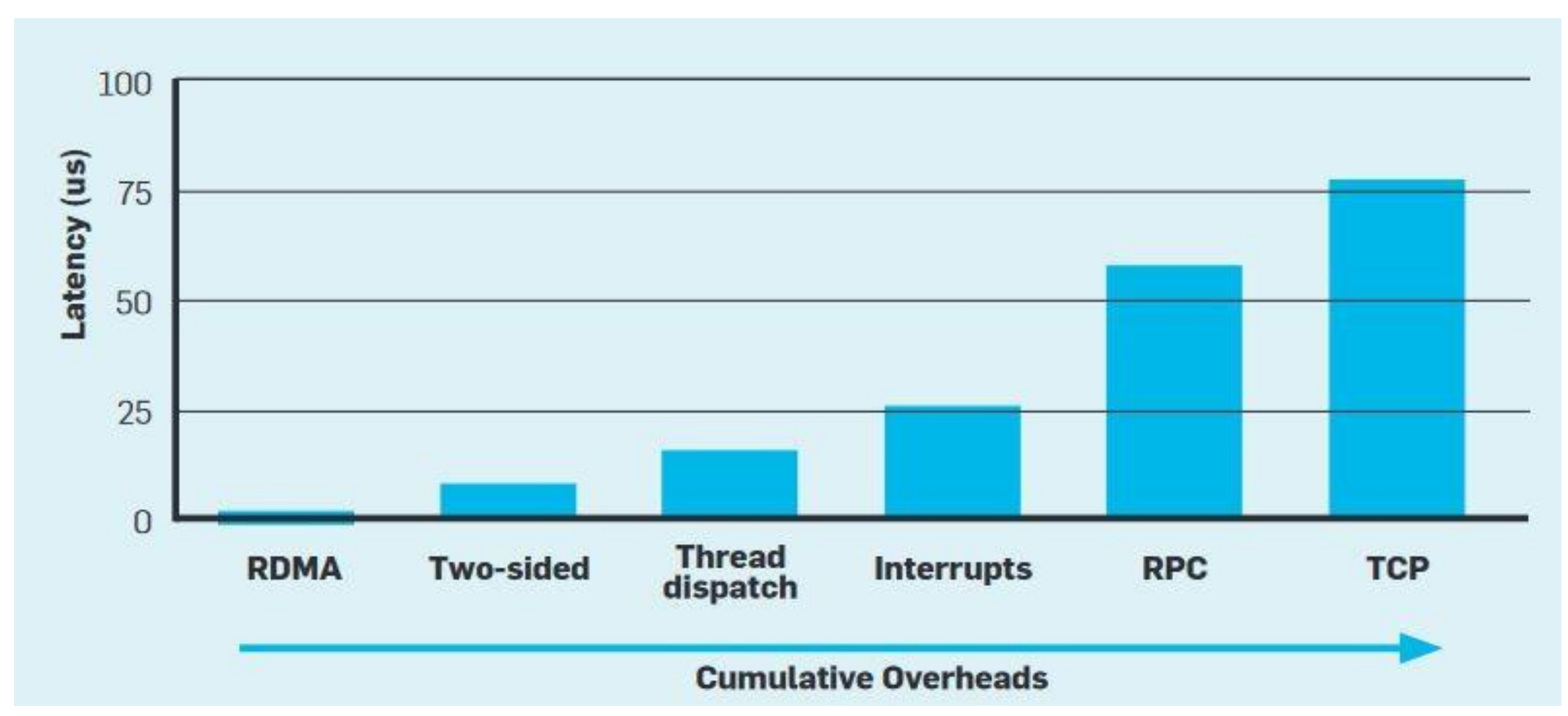
## Asynchronous model: program sends a request to a device

- Continues to run other tasks while waiting
- Must periodically poll or use interrupts to figure out when request is complete

## Can be complex to implement at scale

- Case study: Google datacenter applications with multiple interacting systems in multiple languages touched by thousands of devs
- Changing a Google DC app from an async to sync model resulted in:
  - Improved performance
  - Simpler and easier to understand code

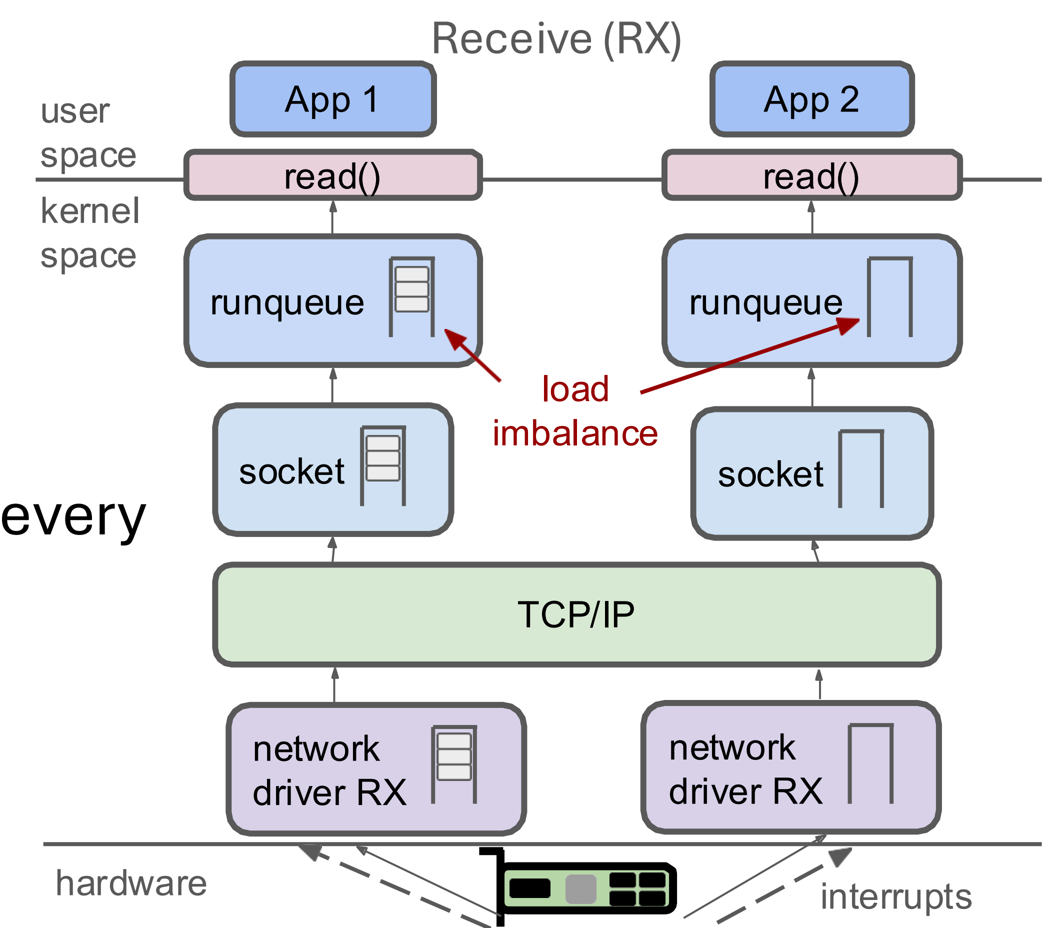
# How to Waste a Fancy, expensive NIC



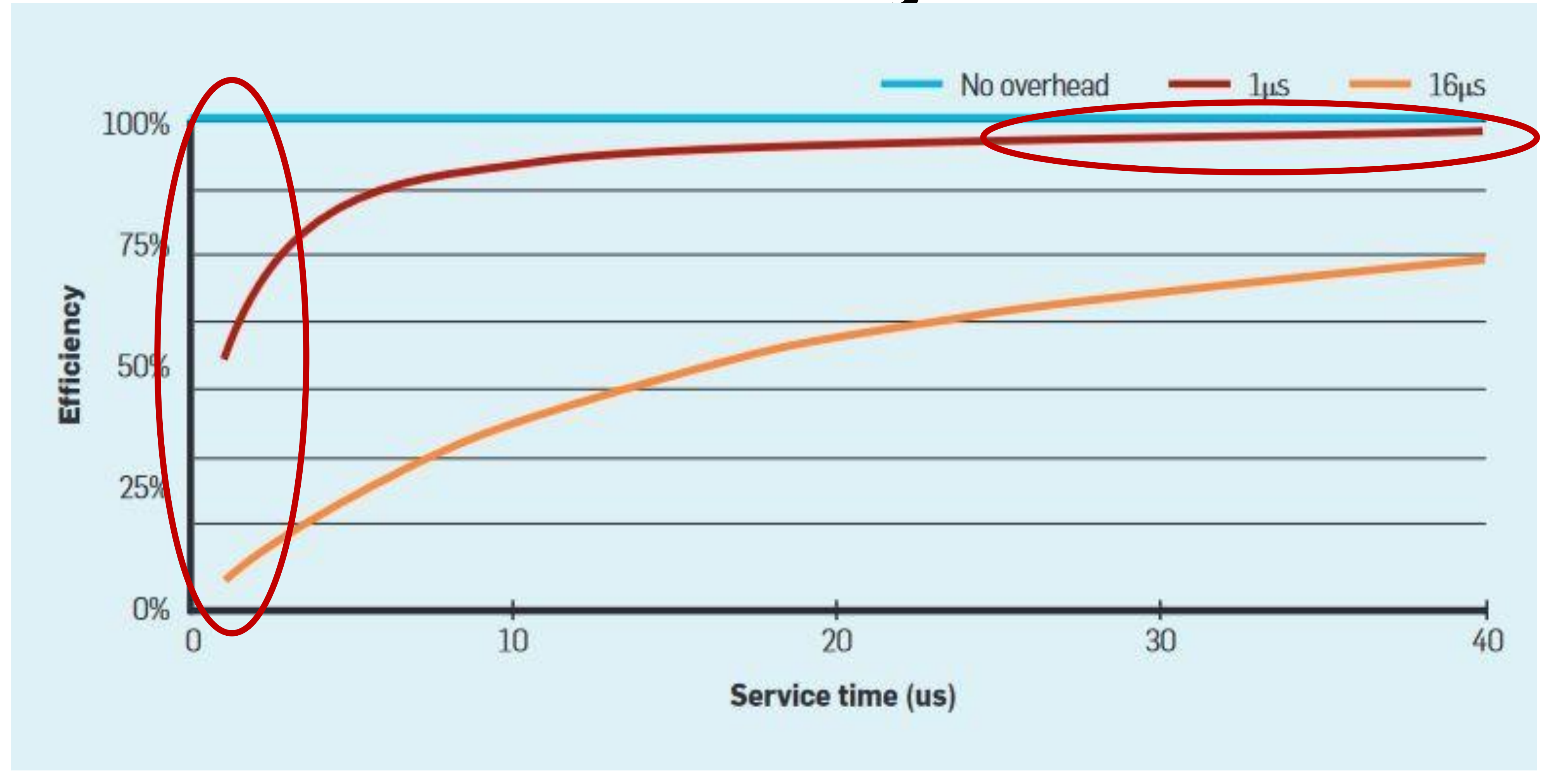
# How does the OS Add So Much Overhead?

## Focus on the receive path

- Multicore example
- Sources of overhead:
  - Context switches
  - Lots of queueing
  - Copies
  - Load imbalance (balances run queues every 4 ms)
  - Packets can arrive at the wrong core
  - Applications can interrupt each other



# Does This Really Matter?



# Yet Another Source of Overhead

**“Datacenter Tax”: Tasks that result in computation that must span machines**

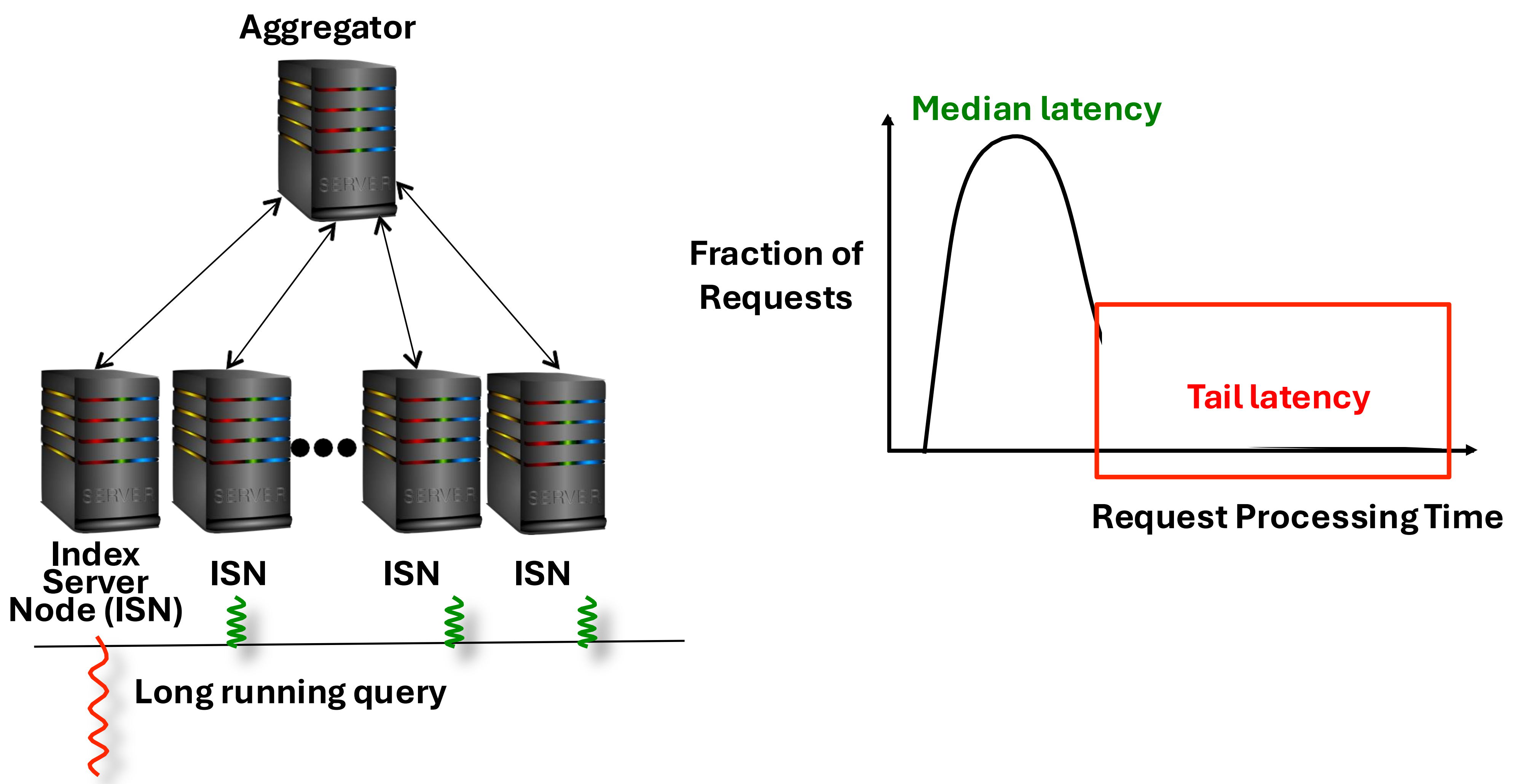
- Serialization/Deserialization of data
- Memory allocation and deallocation
- Network stack costs
- Compression
- Encryption

**Between 20-25% of processor cycles are spent on these tasks**

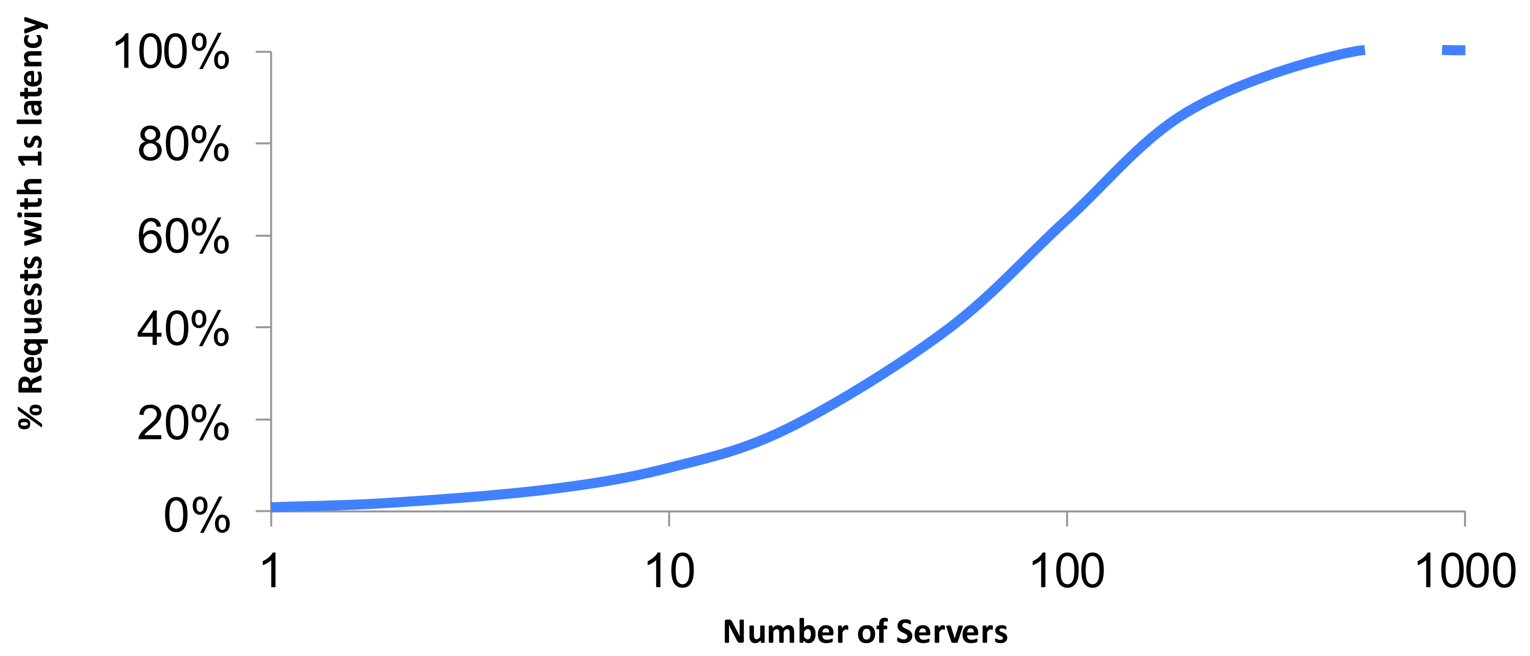
# Another Problem for Datacenters: Tail Latency

***Process TBs of data with O(ms) request latency***

# Tail Latency and its Significance



# The tail at Scale




**Latency > 1s:**

- **63%** of requests at 100--node scale
- **99%** at 500--node scale

# Another Problem for Datacenters: Tail Latency

**Datacenter applications should appear fast and interactive**

- E.g., search auto-completion, snappy search results

**Temporary high latency episodes can degrade responsiveness at large scale**

**Rare spikes in latency can affect a significant portion of requests**

- Why/how?

# What Causes Latency Variability?

**Shared resources leading to contention**

- And global resource sharing (e.g., network switches, shared FS)

**Background daemons and maintenance activities (e.g., garbage collection)**

**Queueing at many layers**

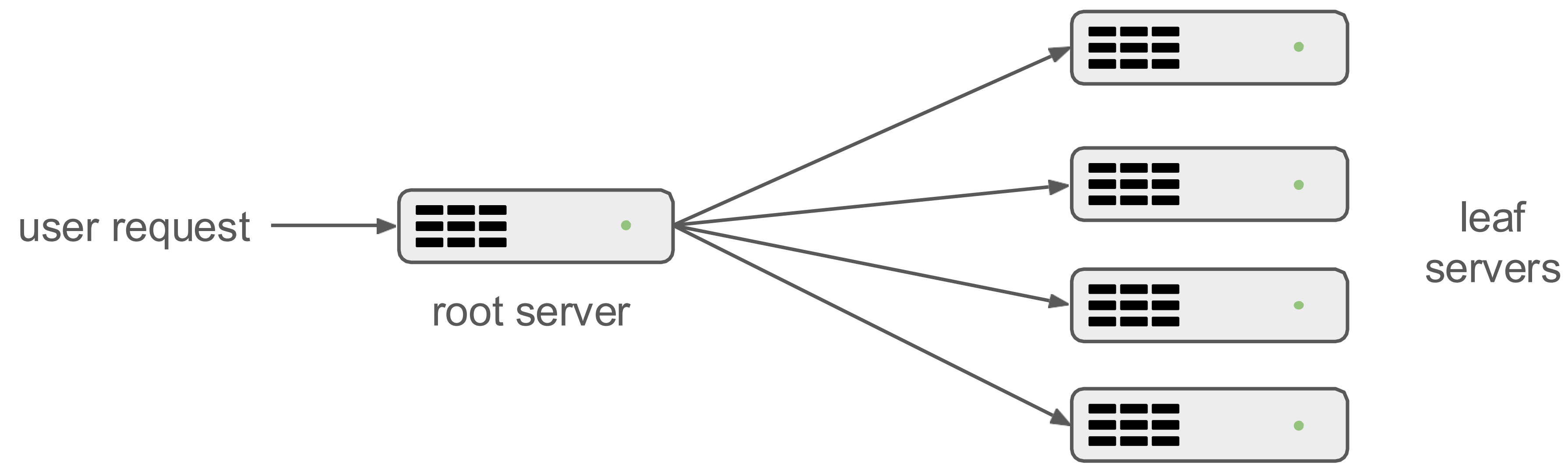
**Power limits and energy management**

# Parallelization to Reduce Latency

**Break a user request into parallelizable sub-operations**

**Fan out requests from root server to leaf servers**

**Leaf servers perform task, respond back to root server**

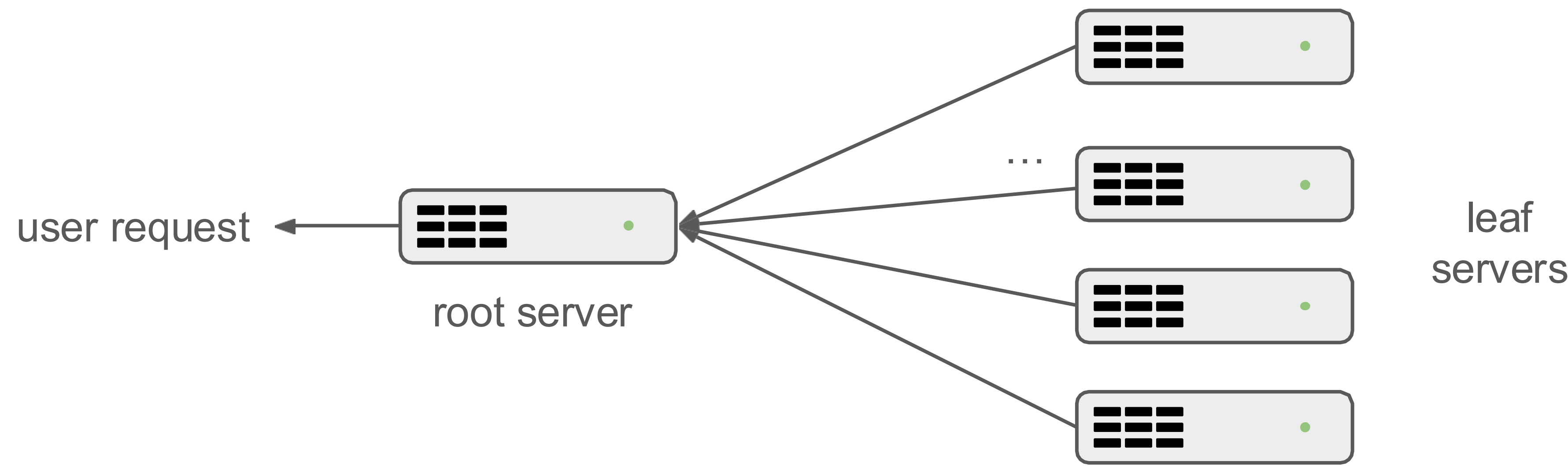


# Parallelization to Reduce Latency

**Break a user request into parallelizable sub-operations**

**Fan out requests from root server to leaf servers**

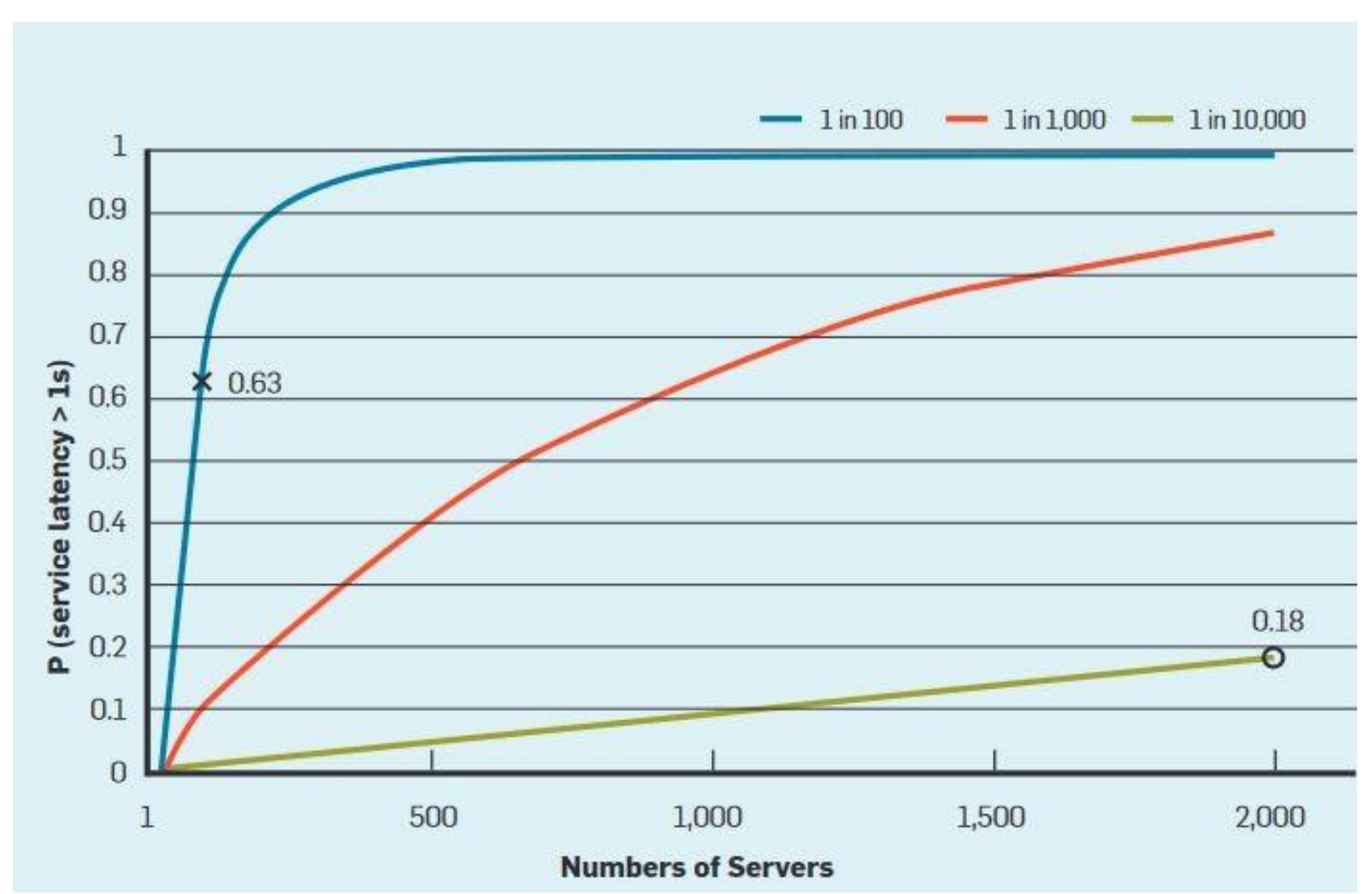
**Leaf servers perform task, respond back to root server**



# Effects of Latency Variability

Large fanouts exacerbate degraded latency from slow servers

Significant lower perf for a large portion of user requests



# How to Reduce Latency Variability

**Separate service classes**

- Separate latency-critical tasks from latency-insensitive/batch tasks

**Reduce head-of-line blocking**

- Break longer requests into shorter requests, time-slice between them

**Manage background activity**

- Careful scheduling of background tasks to reduce interference

**But it's infeasible to eliminate all sources of latency variability…**

# Short Term Adaptions to Latency Variability

**Take advantage of techniques from fault tolerance (e.g., replication)**

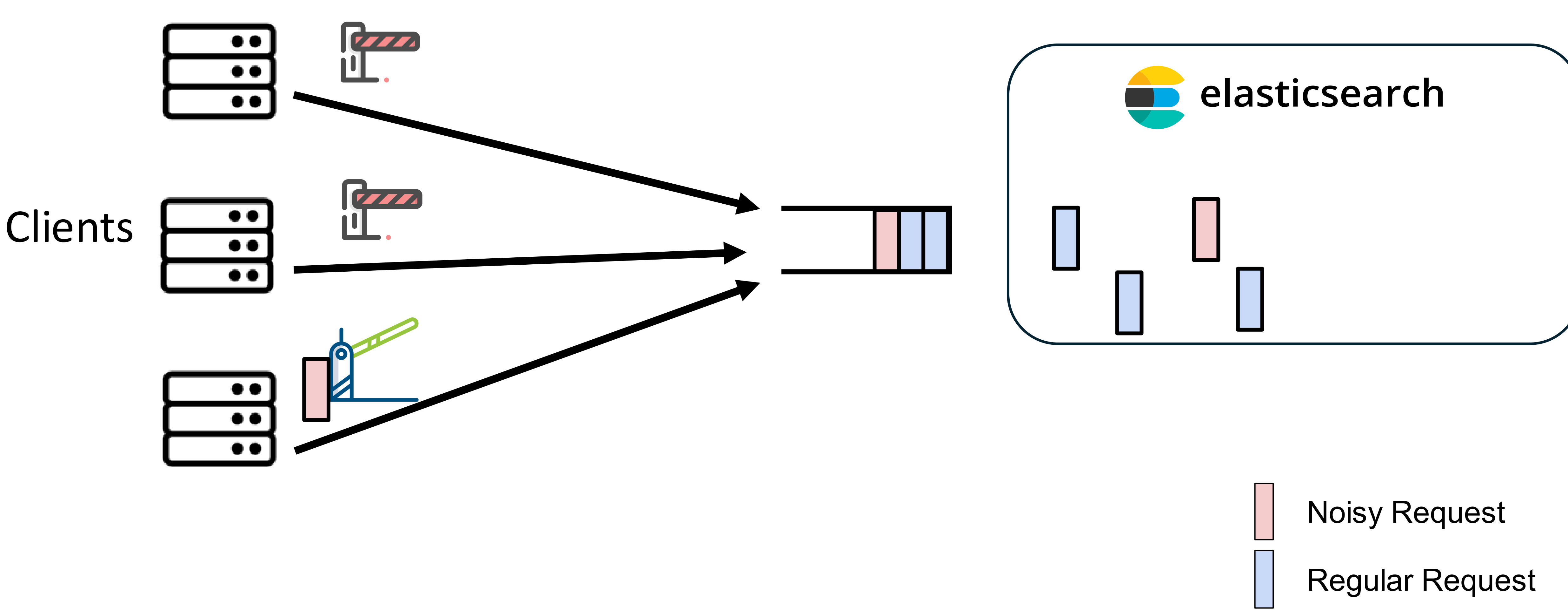
**Hedged requests: Send out request to multiple servers, use first reply**

- Cancel outstanding requests once first result is received
- What if multiple servers execute the same request simultaneously & unnecessarily?
- Must be careful to avoid adding unacceptable extra load – be smart at sending 2nd request

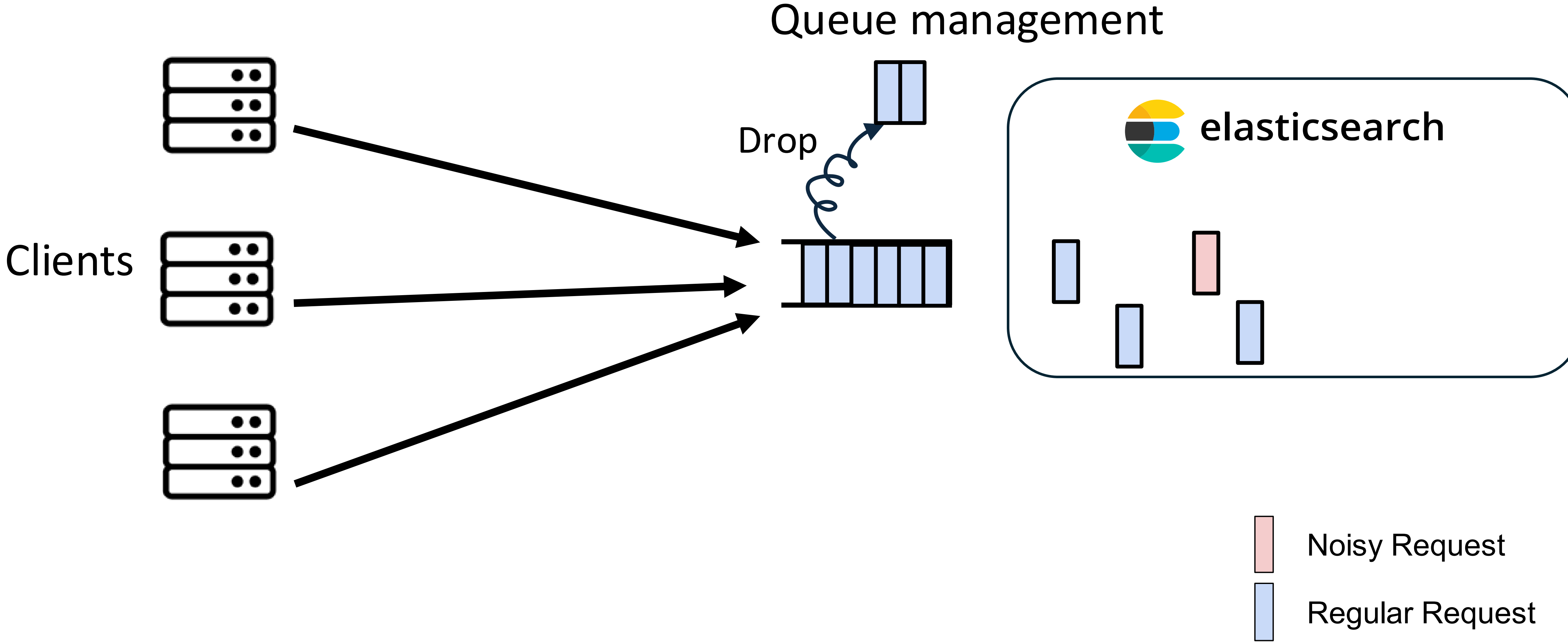
**Tied requests: A main source of latency in lagged requests is queueing**

- Allow a client to choose a server based on queue lengths

# Admission Control



# Queue Management



# Short Term Adaptions to Latency Variability

**Tied requests: A main source of latency in lagged requests is queueing**

- Allow a client to choose a server based on queue lengths
- Enqueue copies of a request at multiple servers simultaneously, allow servers to communicate status to one another
- What about message delays?

# Longer Term Adaptations to Latency Variability

**Deal with coarser-grained phenomena like load imbalance, service variation**

**Micropartitions: # partitions >> # machines**

- Dynamic assignment and load balance of partitions among machines – finer grained

**Selective replication: More replication for popular items**

- Spread the load among more replicas

**Latency-induced Probation: Don't send requests to slow machines**

- Keep probing machine to figure out when it is no longer suffering from slowdowns

# Other Considerations

## Large information retrieval

- "Good enough" results: Don't wait for every last response, send reply when enough have responded to achieve a "good enough" result
- Canary queries: Don't send a potentially dangerous query to everyone at the same time. Send to one and observe behavior; if safe, send to everyone else

## Mutations

- Latency variation in state updates is usually not much of a concern
- Updates are infrequent, off the critical path, and already latency tolerant

## Hardware trends

- Hardware variability is likely to increase
- Device heterogeneity and increasing scale make software tolerance even more important
- Better hardware can also make latency tolerance cheaper

# Performance in Datacenters Is Hard

**Faster hardware increases the importance of low-overhead techniques**

- We don’t want software wasting our fancy new hardware
- Need to redesign software stacks with microsecond-scale in mind

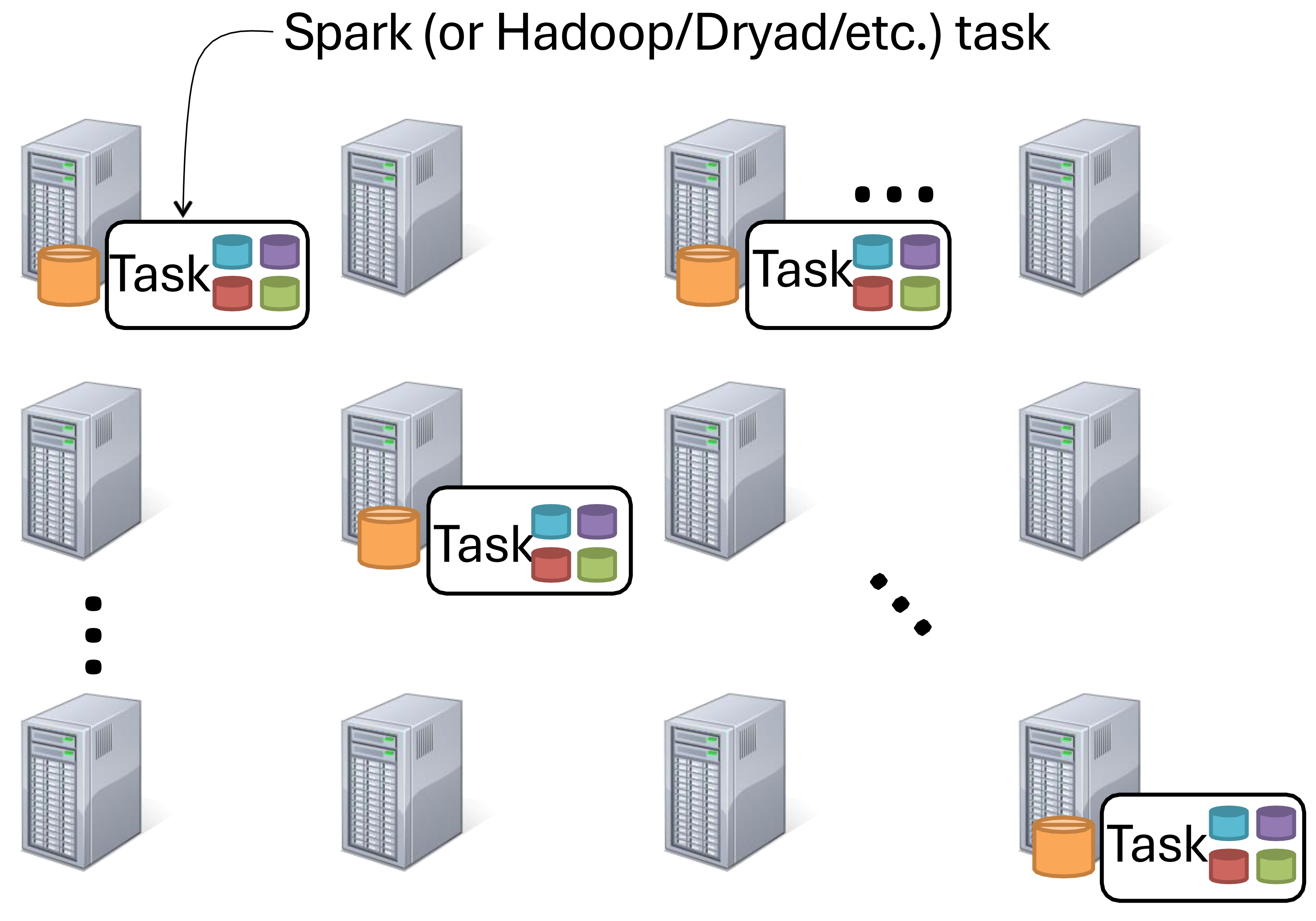
**Latency variation can lead to unacceptable violations in performance**

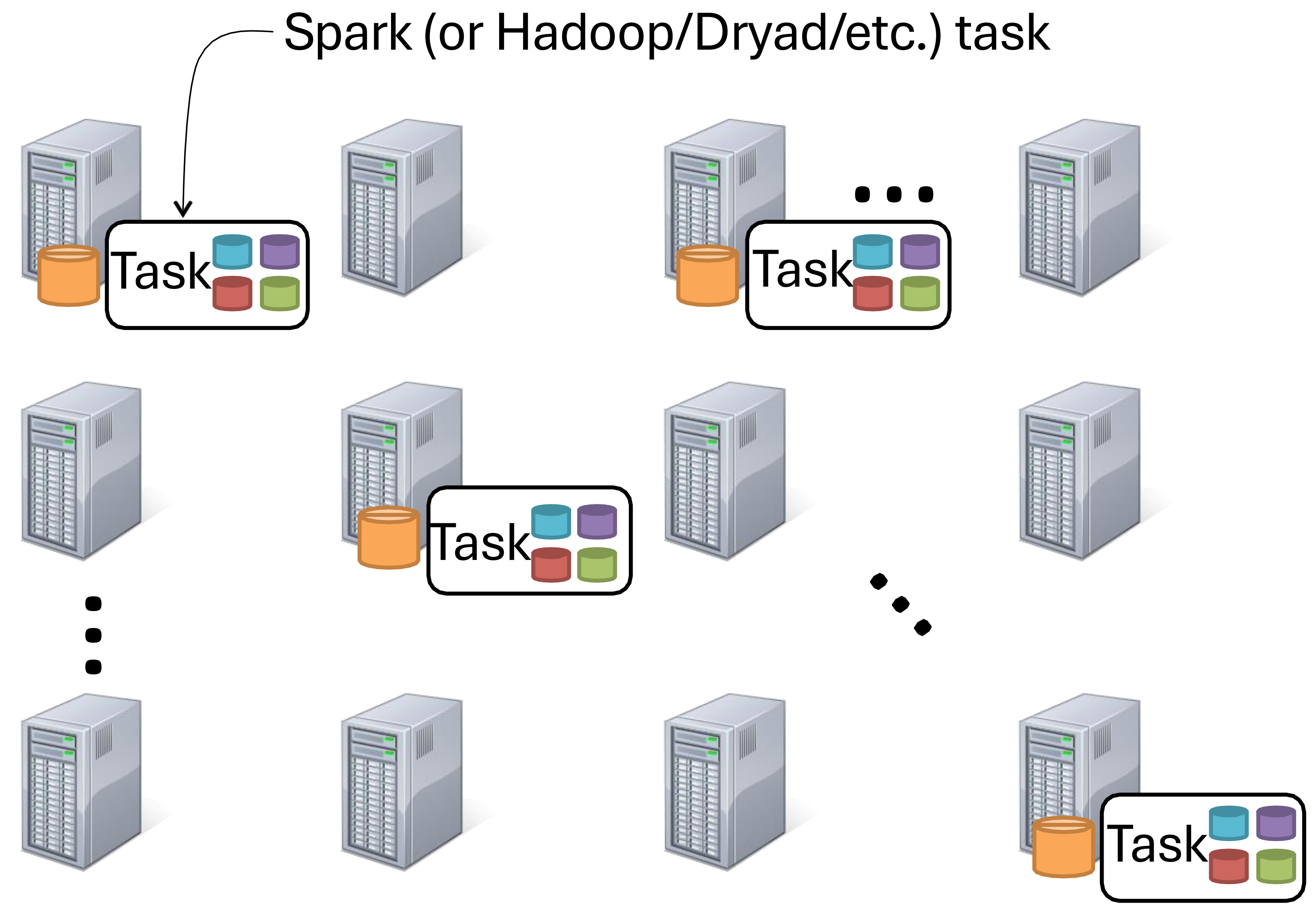
- Fan-out techniques common in datacenters exacerbate this
- Can take advantage of smart techniques to make applications latency tolerant

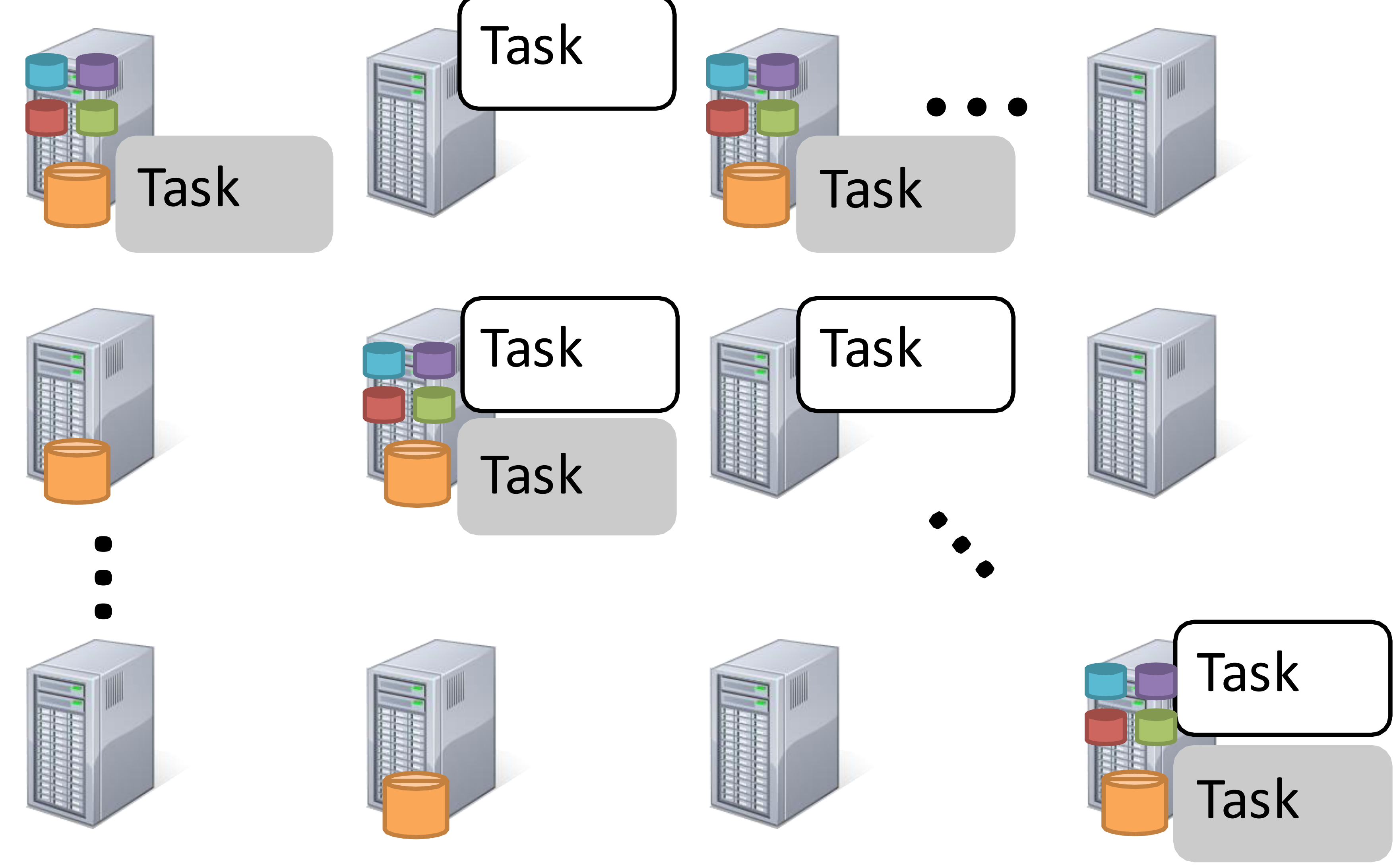
# Making Sense of Performance in Data Analytics Frameworks

Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, Byung-Gon Chun

NSDI'15


Spark (or Hadoop/Dryad/etc.) task
Task
Task
Task
Task


Spark (or Hadoop/Dryad/etc.) task
Task
Task
Task
Task


Task
Task
Task
Task
Task
Task
Task
Task
Task

# Widely-accepted Mantras

**Network and disk I/O are bottlenecks**

**Stragglers are a major issue with unknown causes**

# This Work

(1) How can we quantify performance bottlenecks?

**Blocked time analysis**

(2) Do the mantras hold?

**Takeaways based on three workloads run with Spark**

# Takeaways Based on Three Spark Workloads

**Network optimizations** can reduce job completion time by **at most 2%**

**CPU (not I/O) often the bottleneck** <19% reduction in completion time from optimizing disk

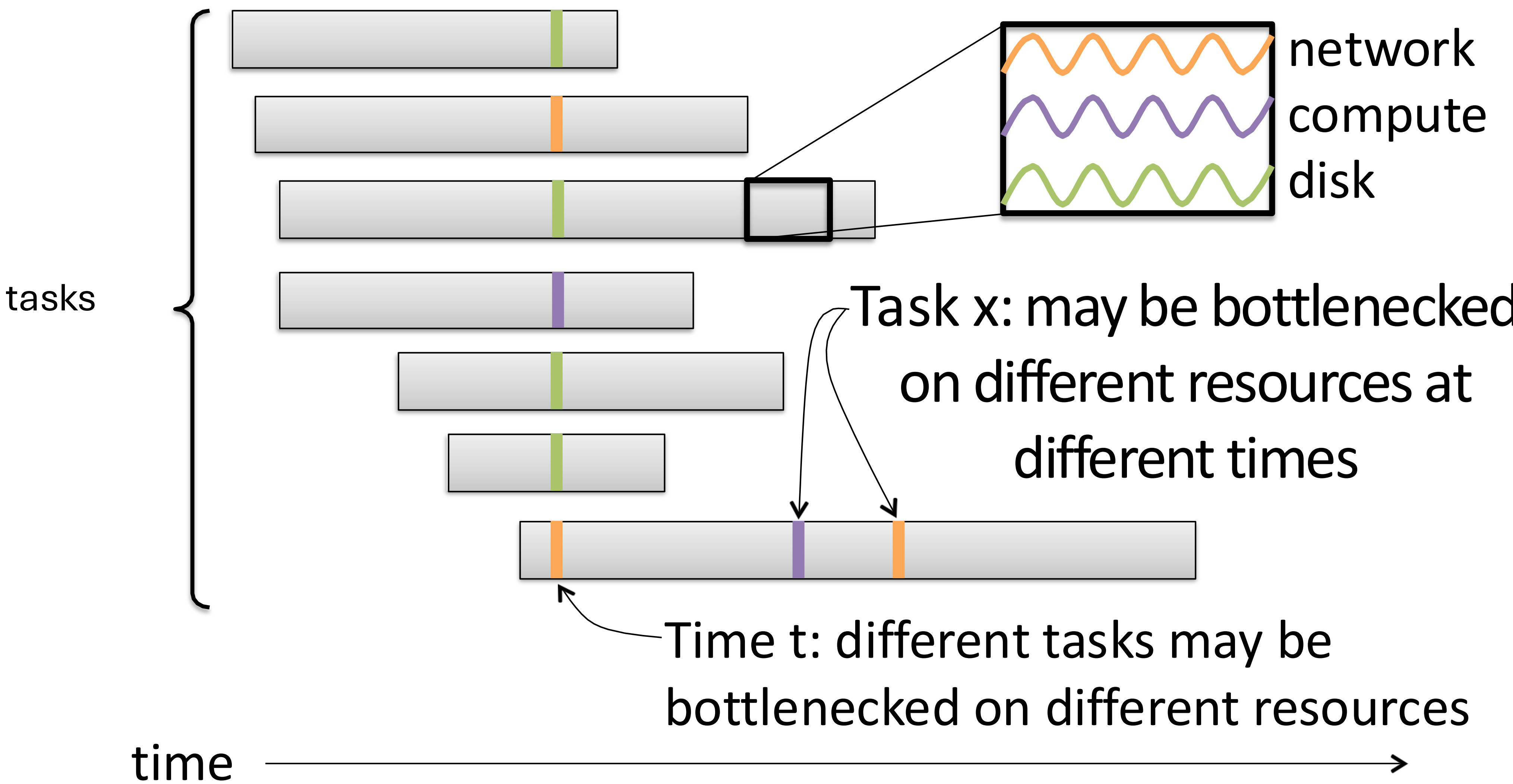
**Many straggler causes can be identified and fixed**

# This Work

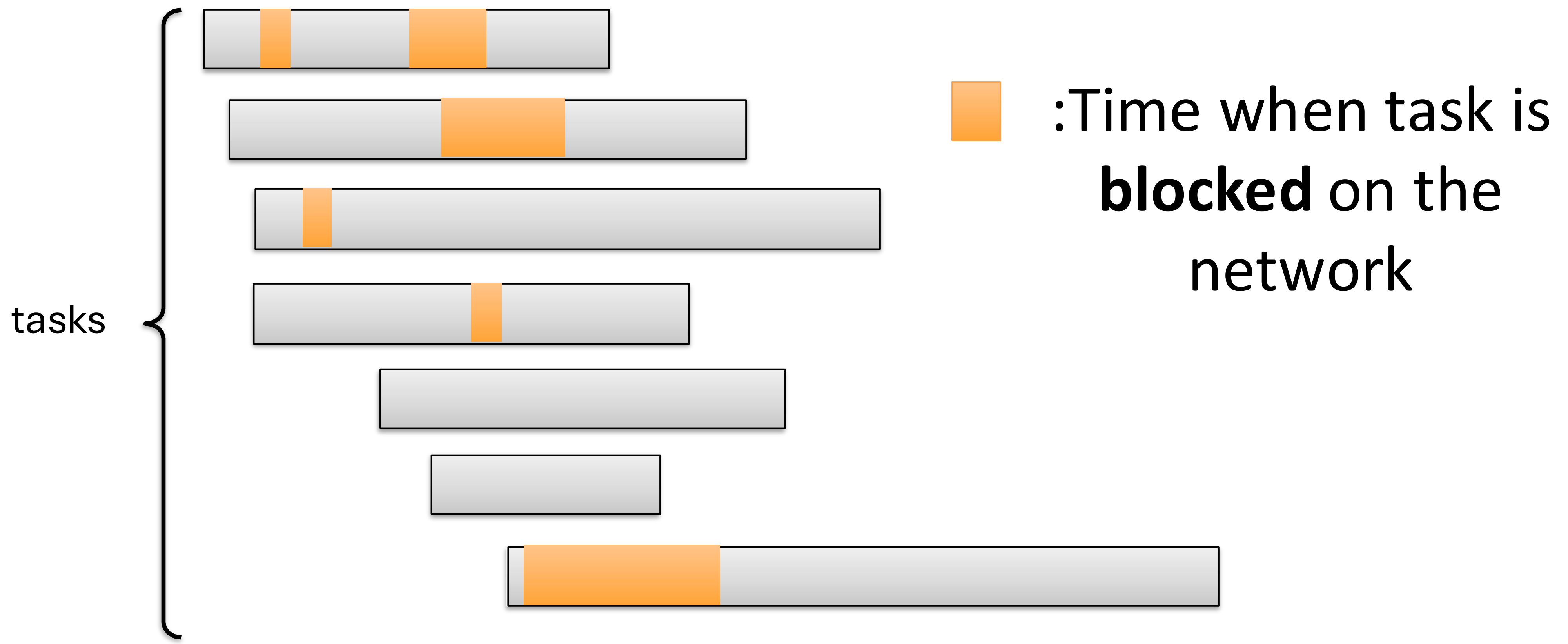
**Accepted Mantras are often not true**

**Methodology to avoid performance misunderstanding in the future**

# What is the Job's Bottleneck?

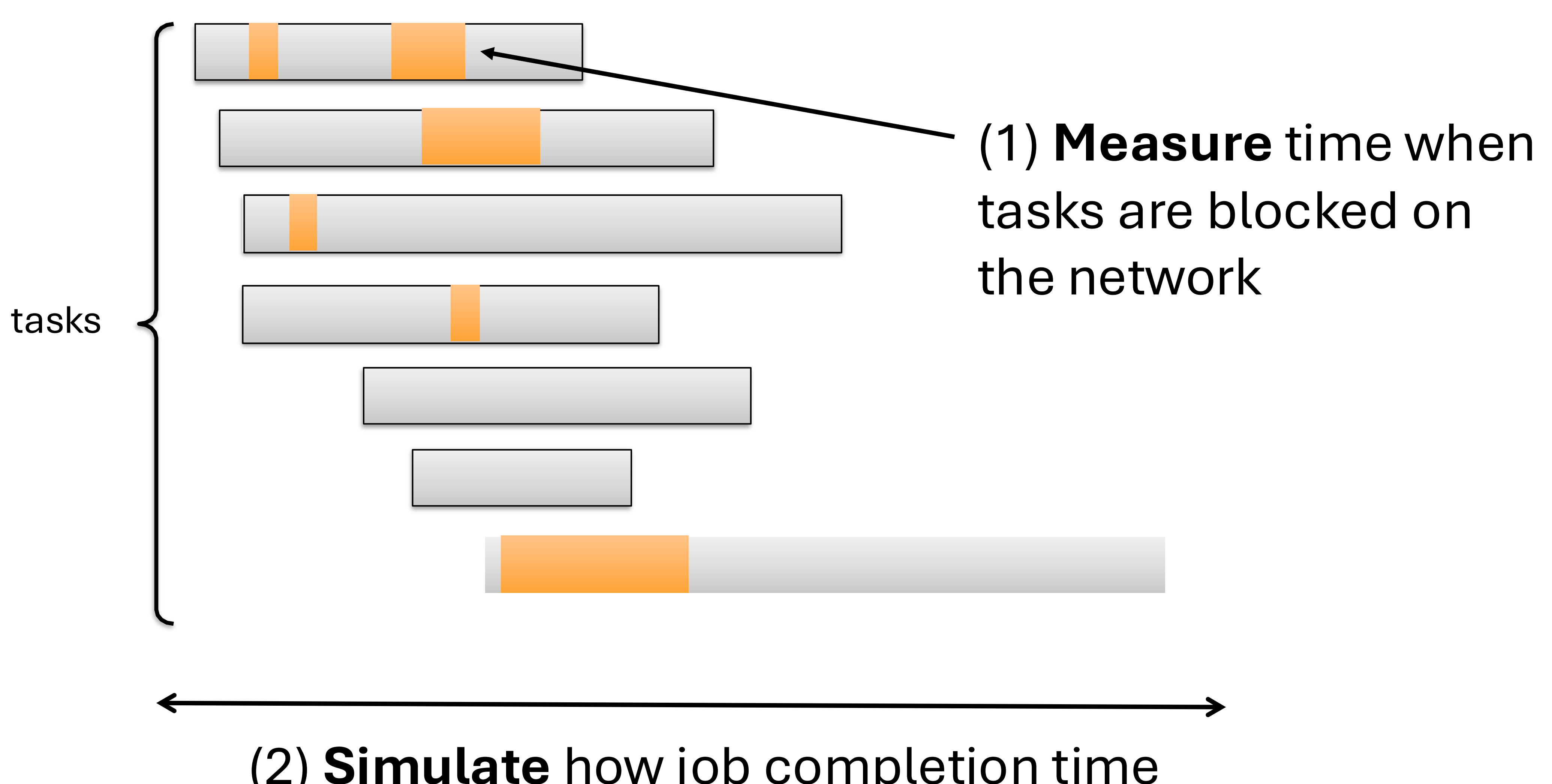


# How does Network Affect the Job's Completion Time?

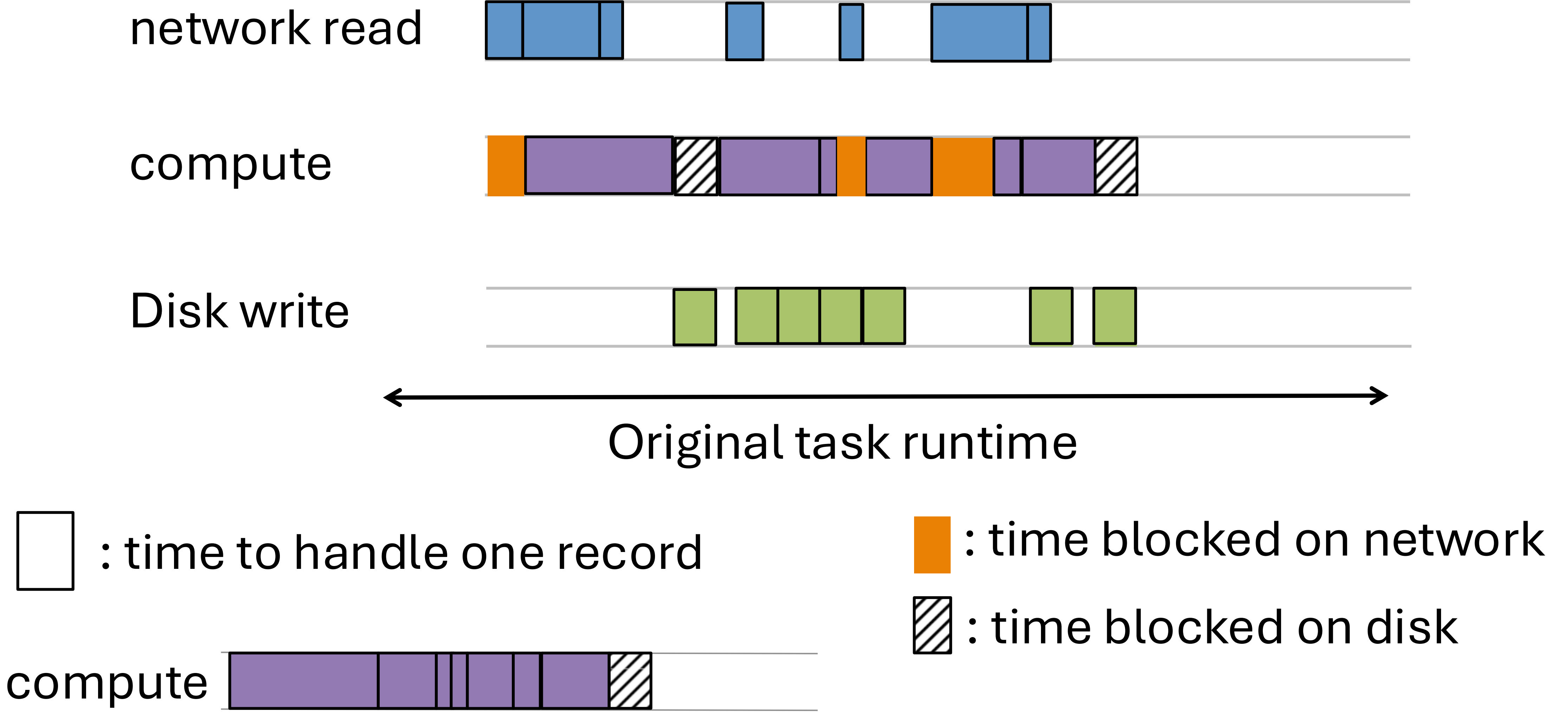



**Blocked time analysis**: how much faster would the job complete if tasks never blocked on the network?

# Block Time Analysis

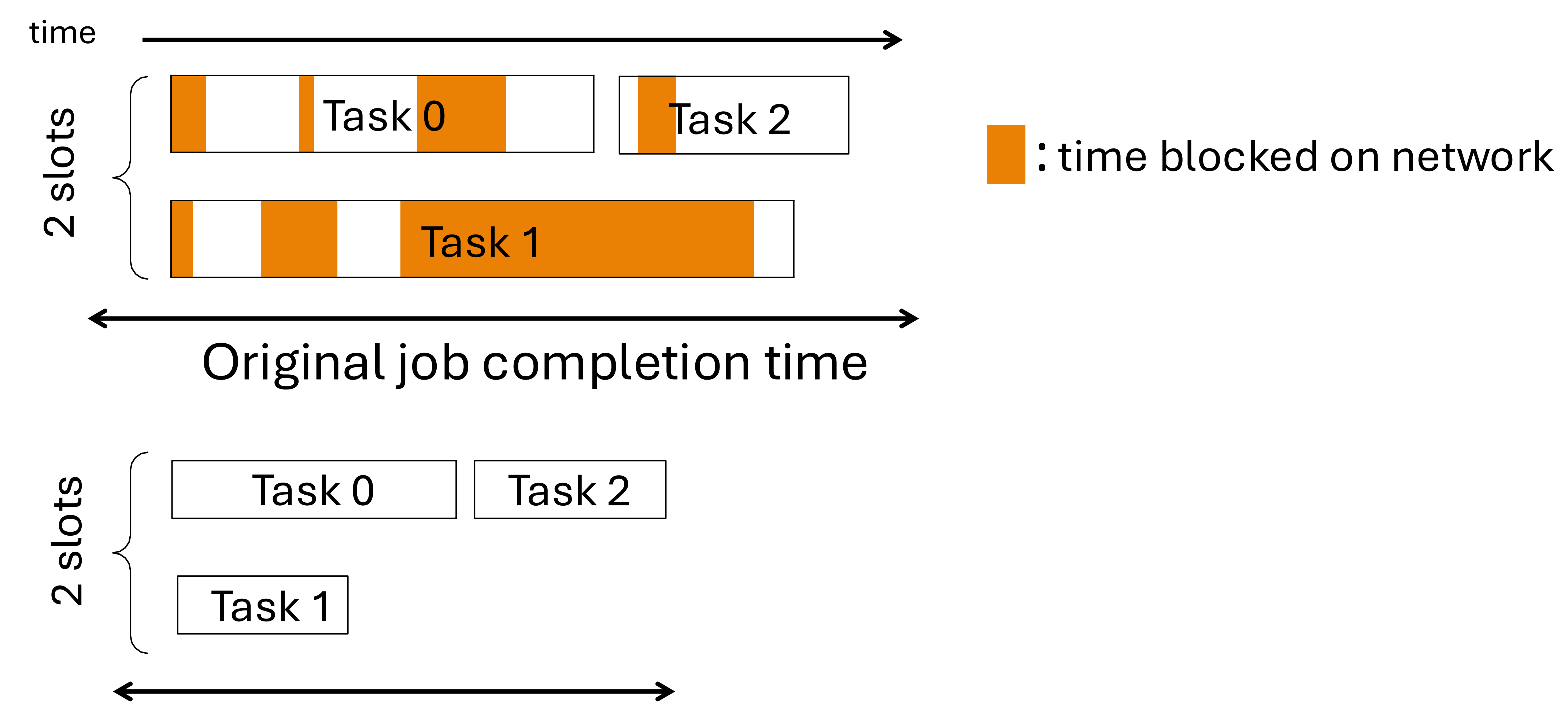


# Measure time when tasks are blocked on network




**Best case** task runtime if network were infinitely fast

# Simulate how job completion time would change



# Quiz Attendance

https://forms.gle/F3L6CPWtEt6vJvdo9

# Quiz

https://forms.gle/WDRM4nPaUM8tJy2g8