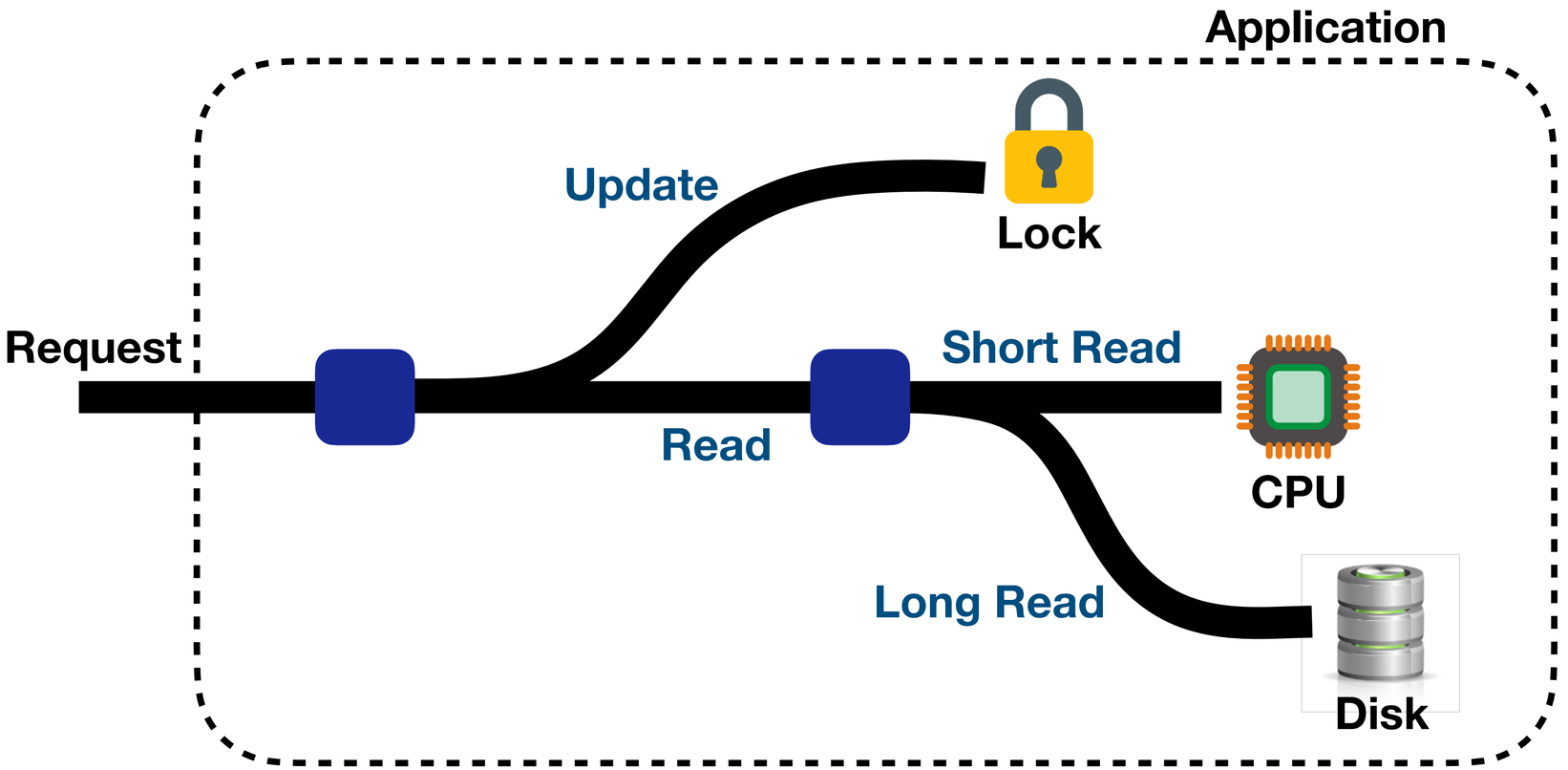
Svalinn: Overload Control in Large-Scale Servers with Multiple Resource Bottlenecks

Bhaskar Pardeshi Peidi Song Ahmed Saeed

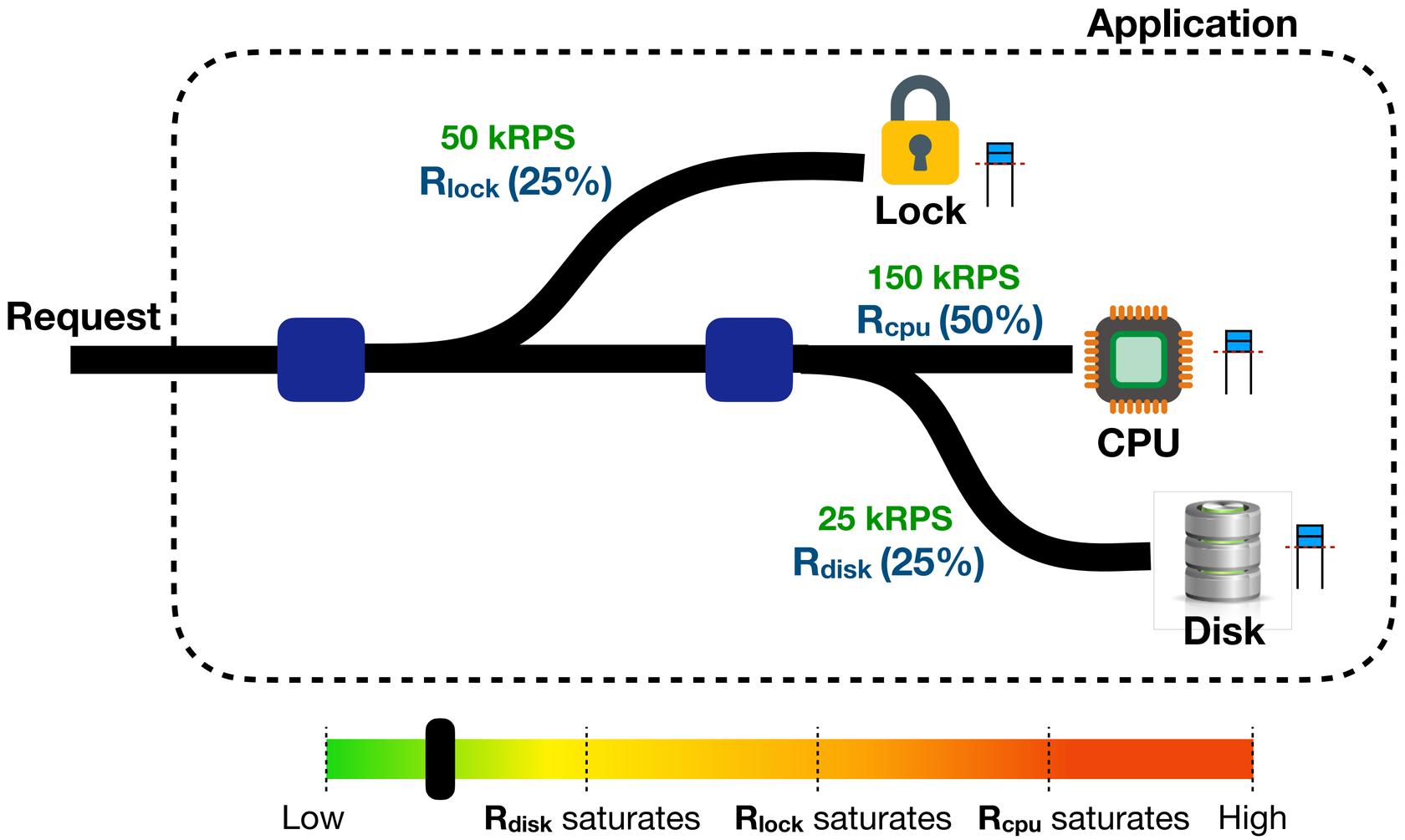


Applications have Heterogenous Bottlenecks

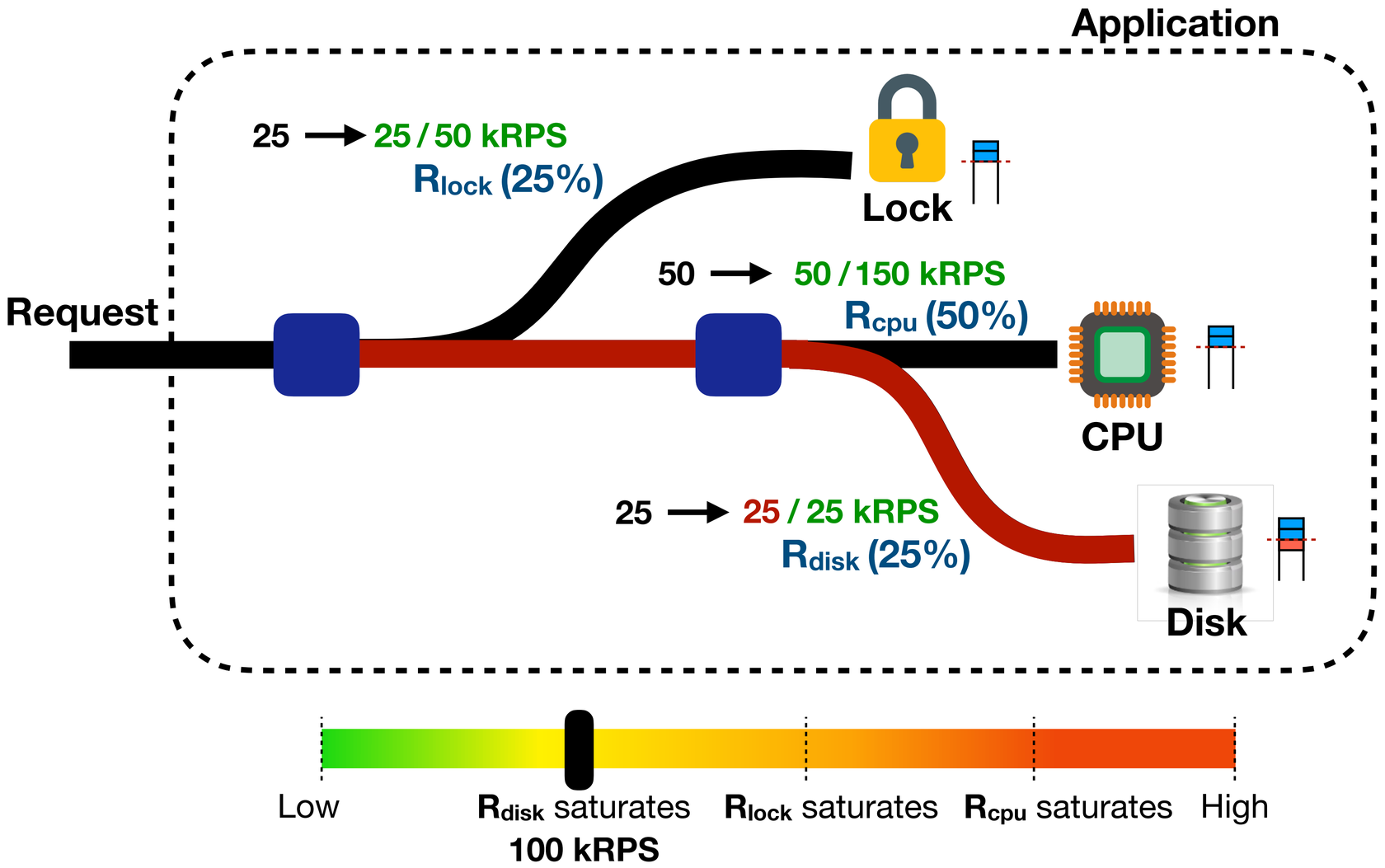


Different requests can bottleneck on different server resources

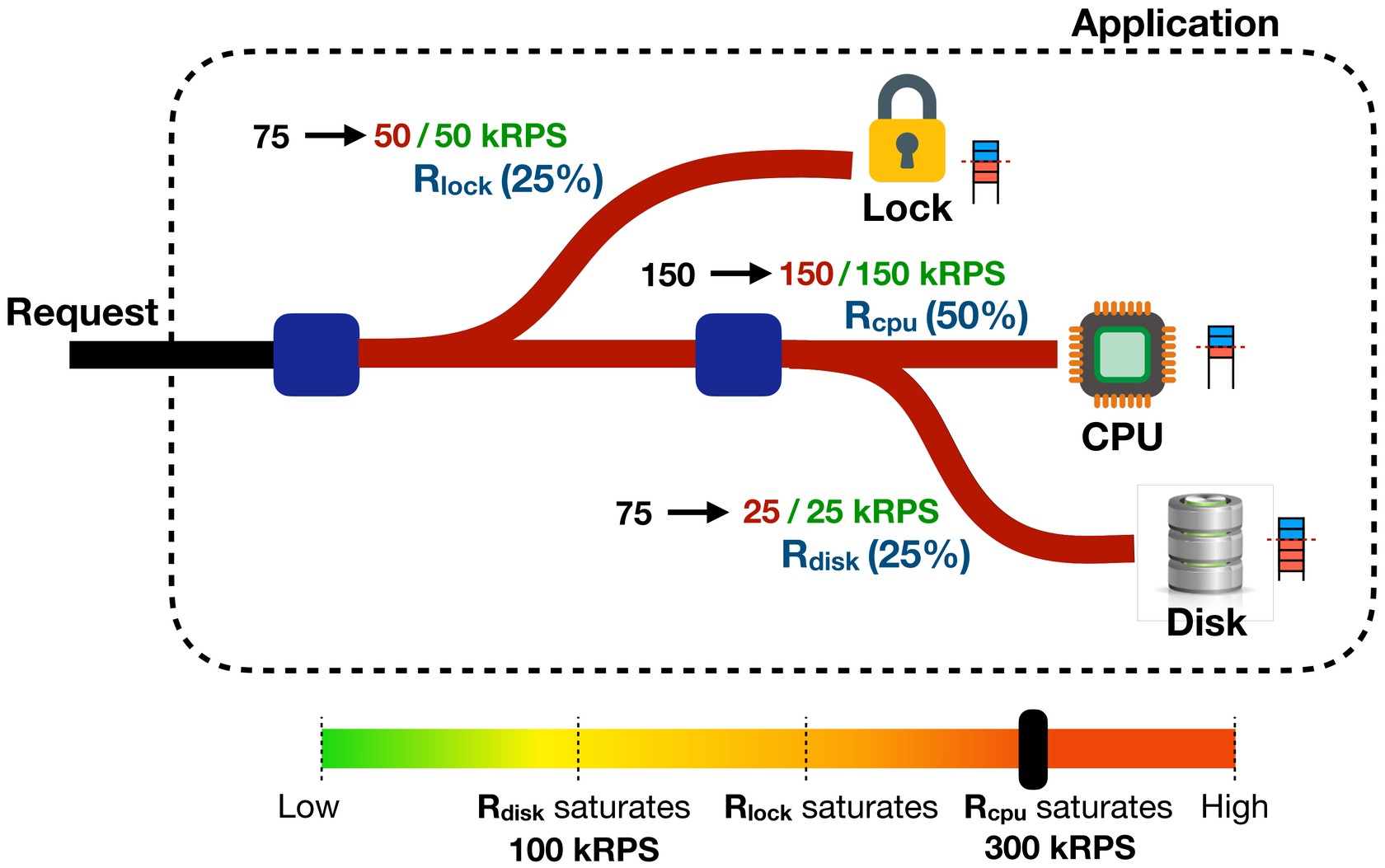
Overload in Multi-Bottleneck Applications



Overload in Multi-Bottleneck Applications

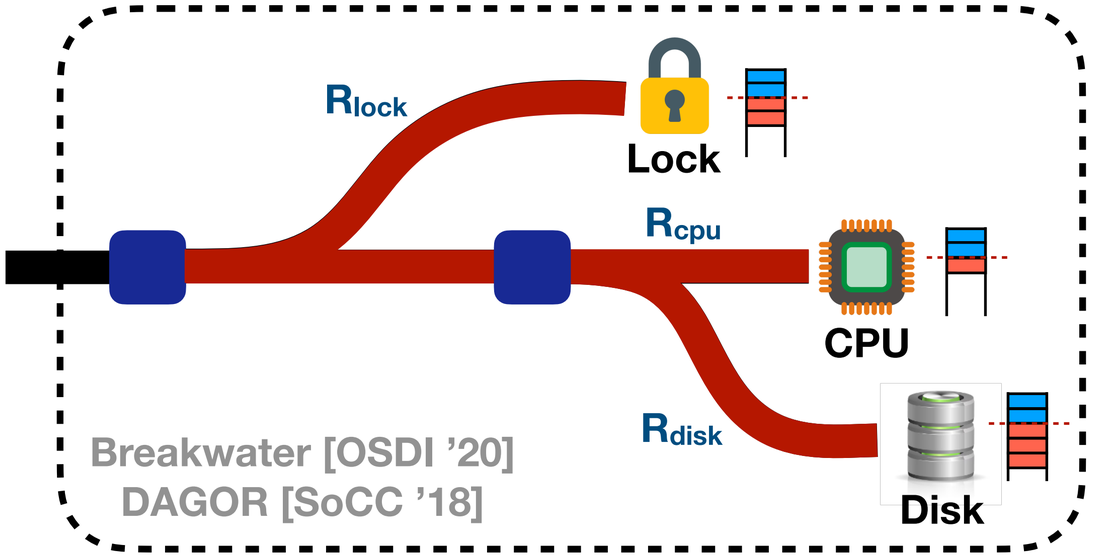


Overload in Multi-Bottleneck Applications



Traditional Overload Controllers

Single-resource queueing delay



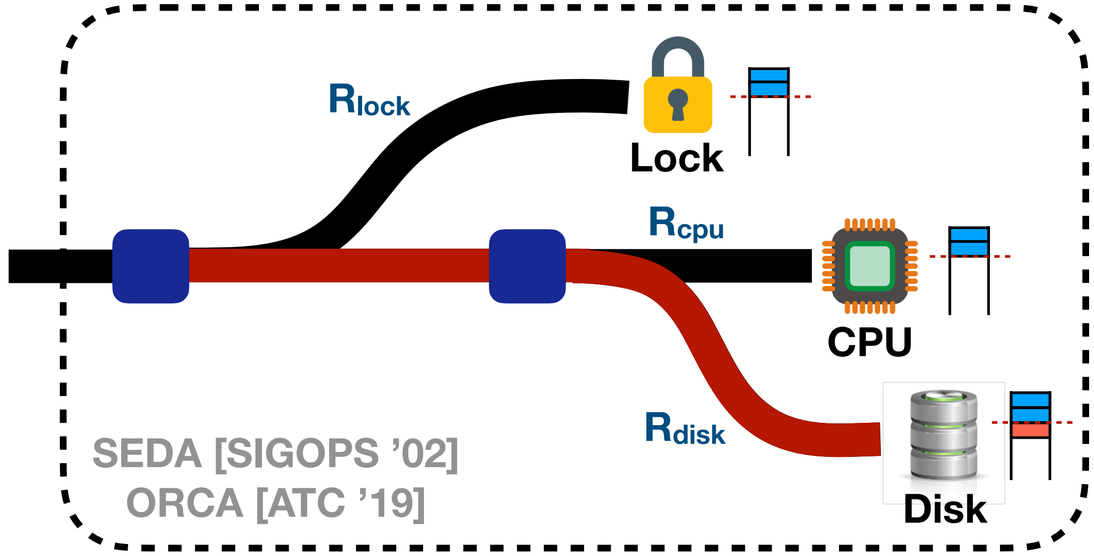


Admitted Load

High Utilization

High Latency

End-to-end delay





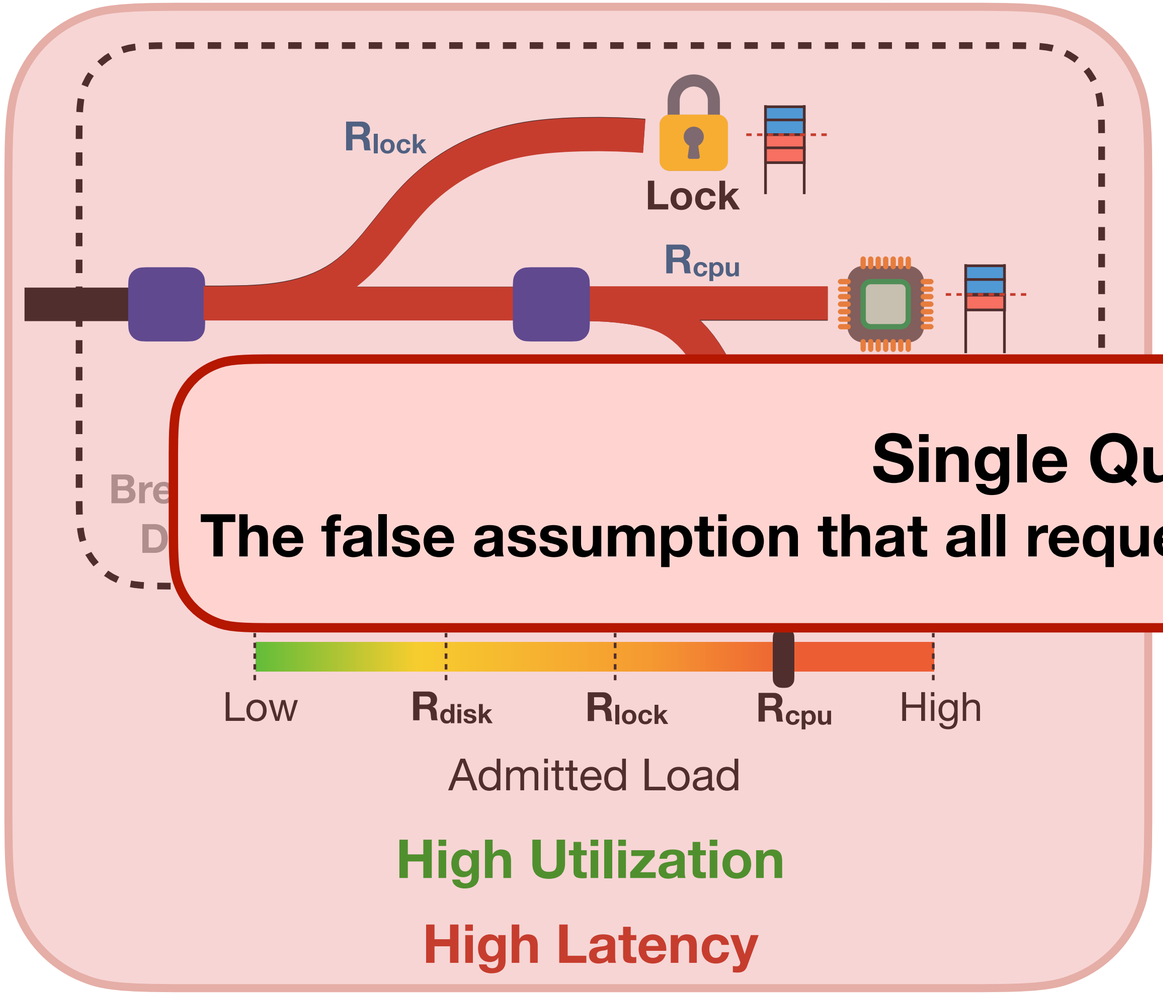
Admitted Load

Low Utilization

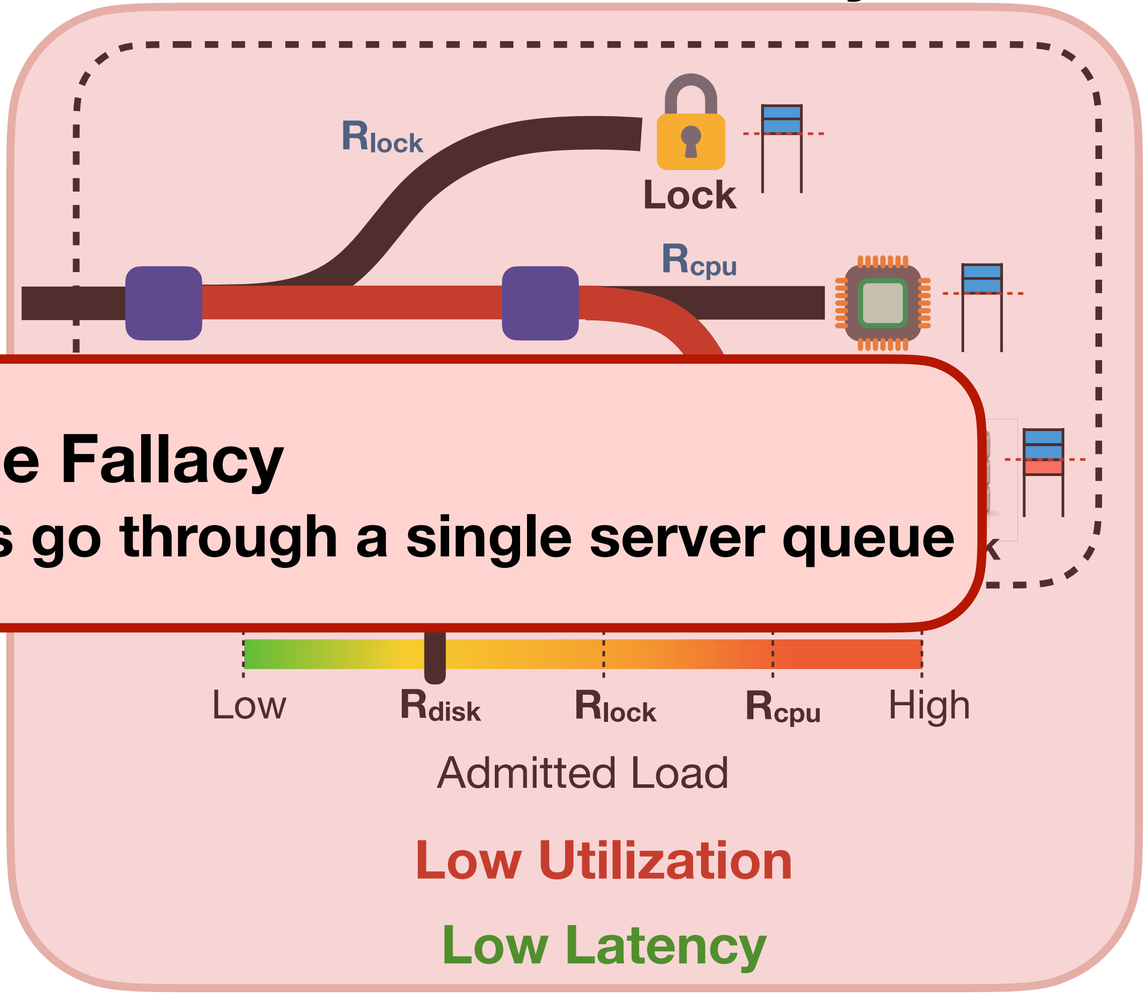
Low Latency

Traditional Overload Controllers

Single-resource queueing delay



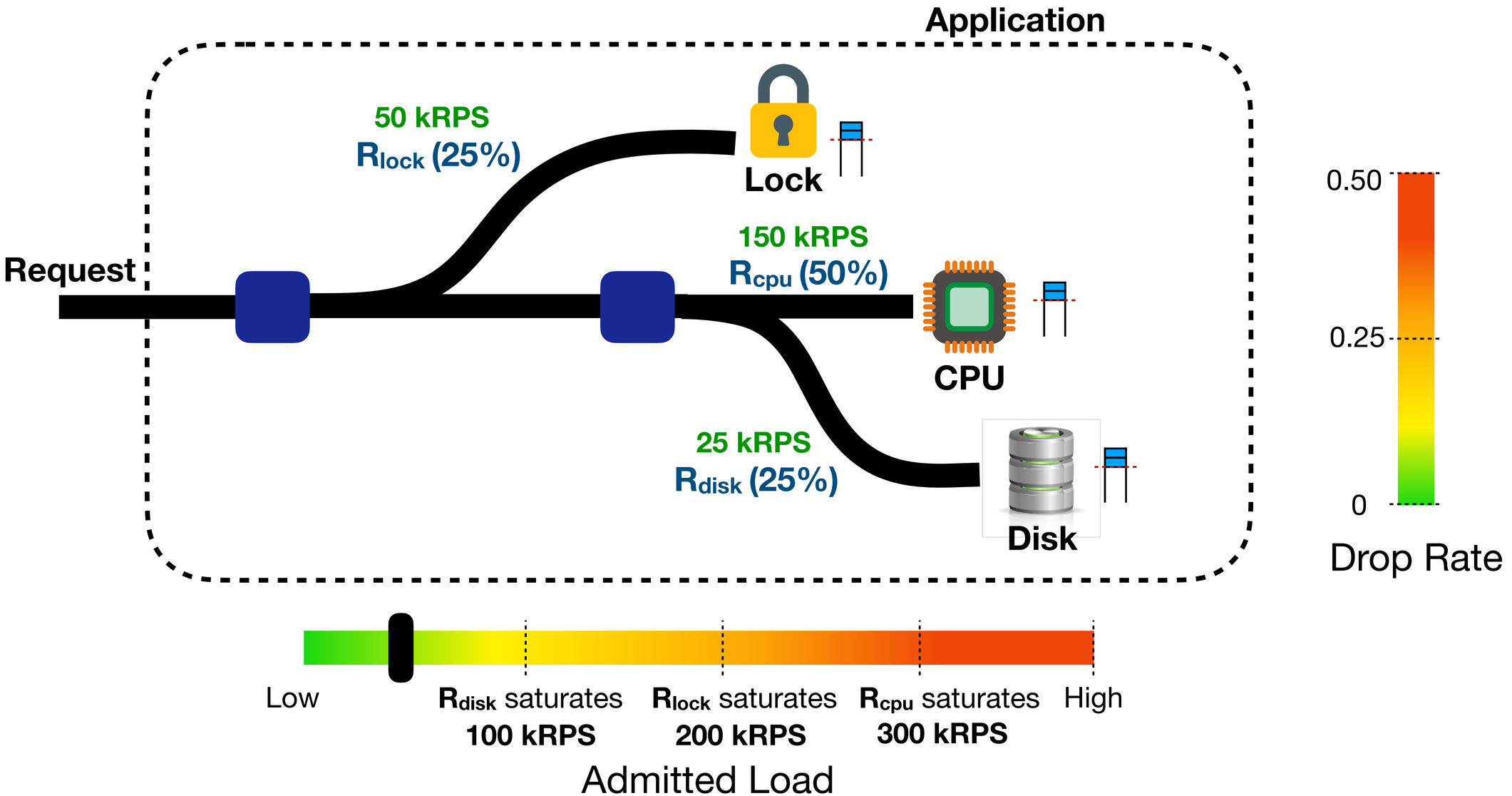
End-to-end delay



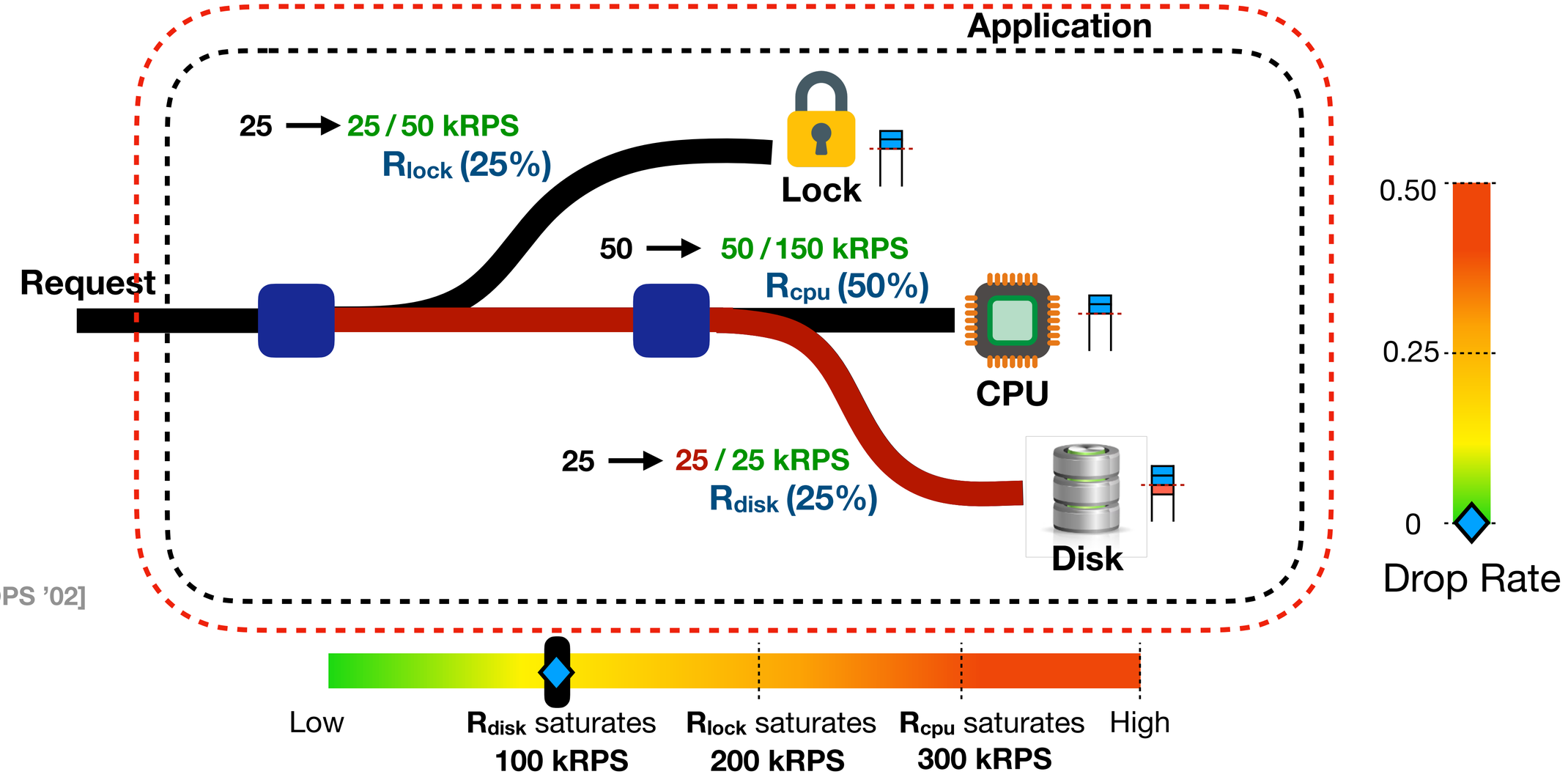
Single Queue Fallacy

The false assumption that all requests go through a single server queue

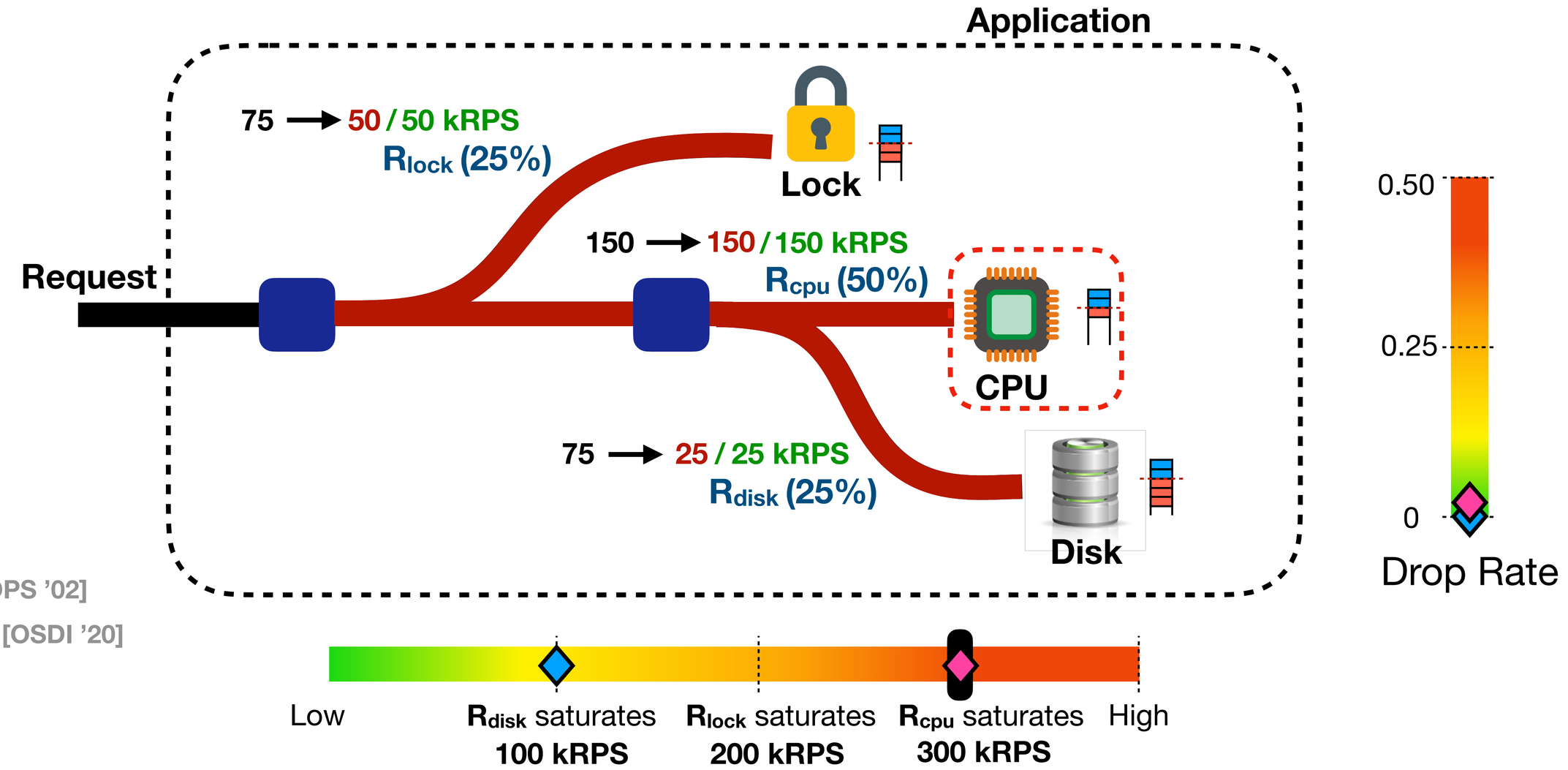
Static Operating Points in the Admission Space



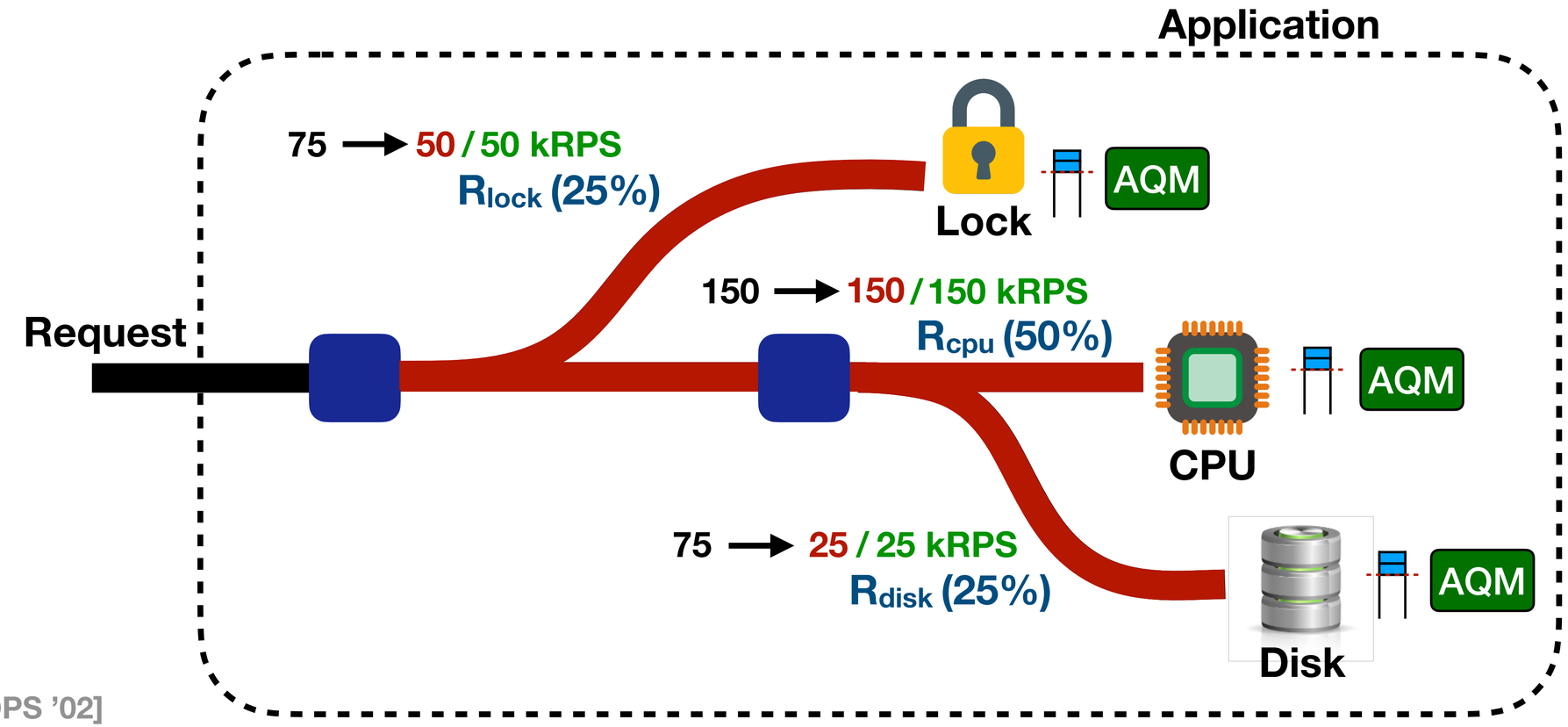
Static Operating Points in the Admission Space

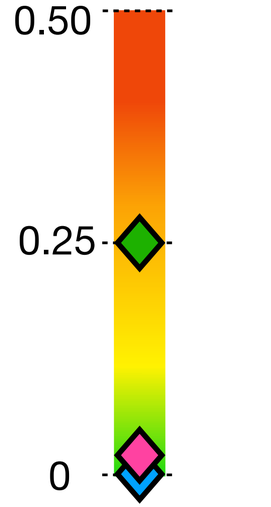


Static Operating Points in the Admission Space



Static Operating Points in the Admission Space

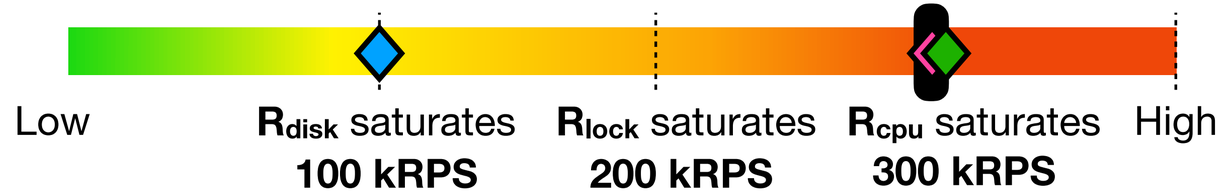




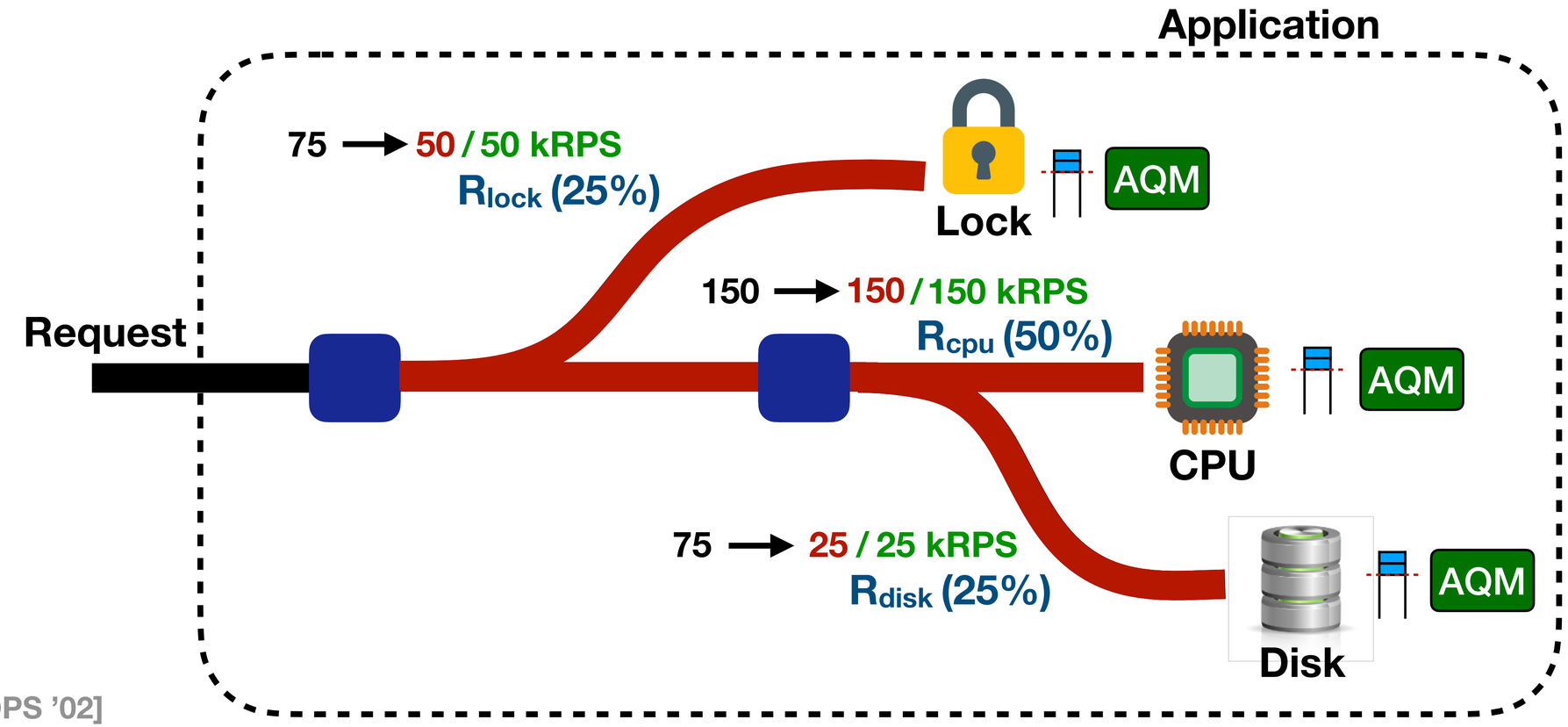
SEDA [SIGOPS '02]

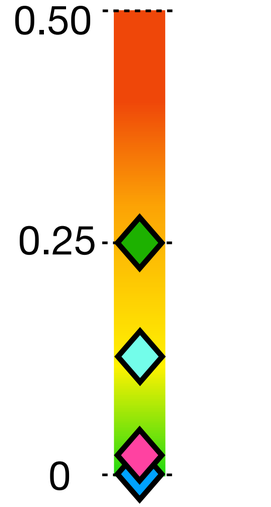
Breakwater [OSDI '20]

Protego [NSDI '23]

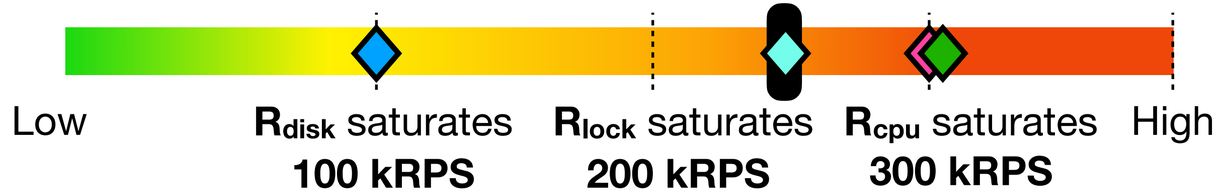


Static Operating Points in the Admission Space

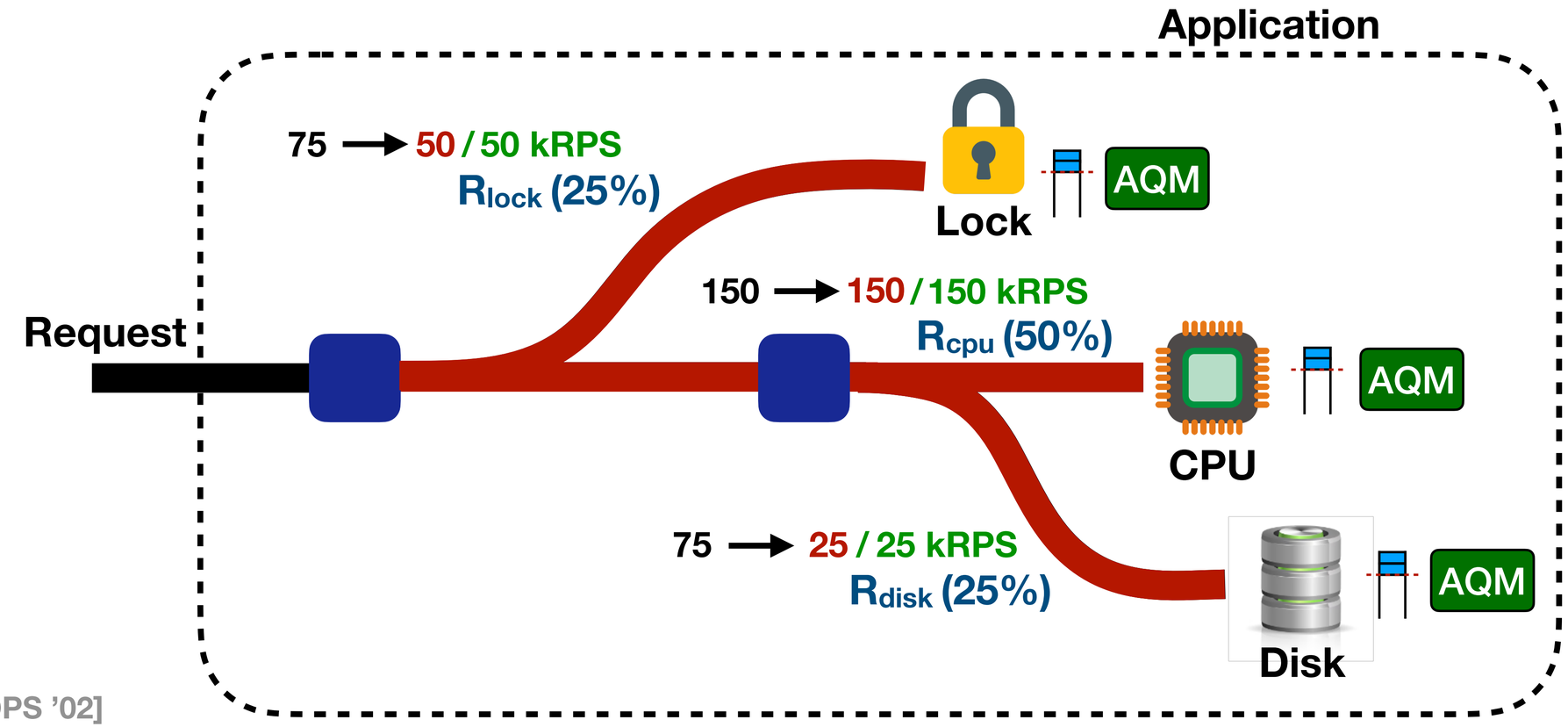


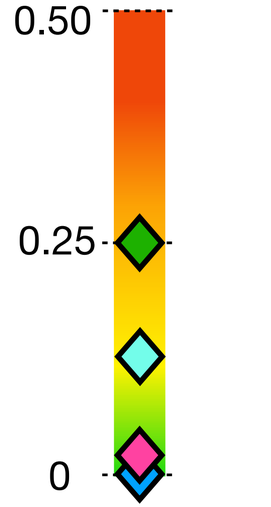


- SEDA [SIGOPS '02]
- Breakwater [OSDI '20]
- Protego [NSDI '23]
- Desired admission point

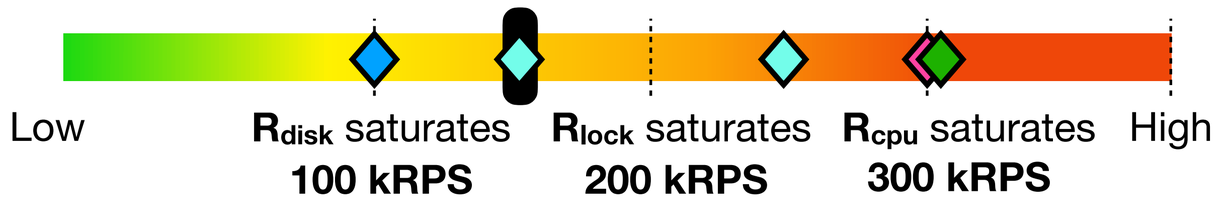


Static Operating Points in the Admission Space

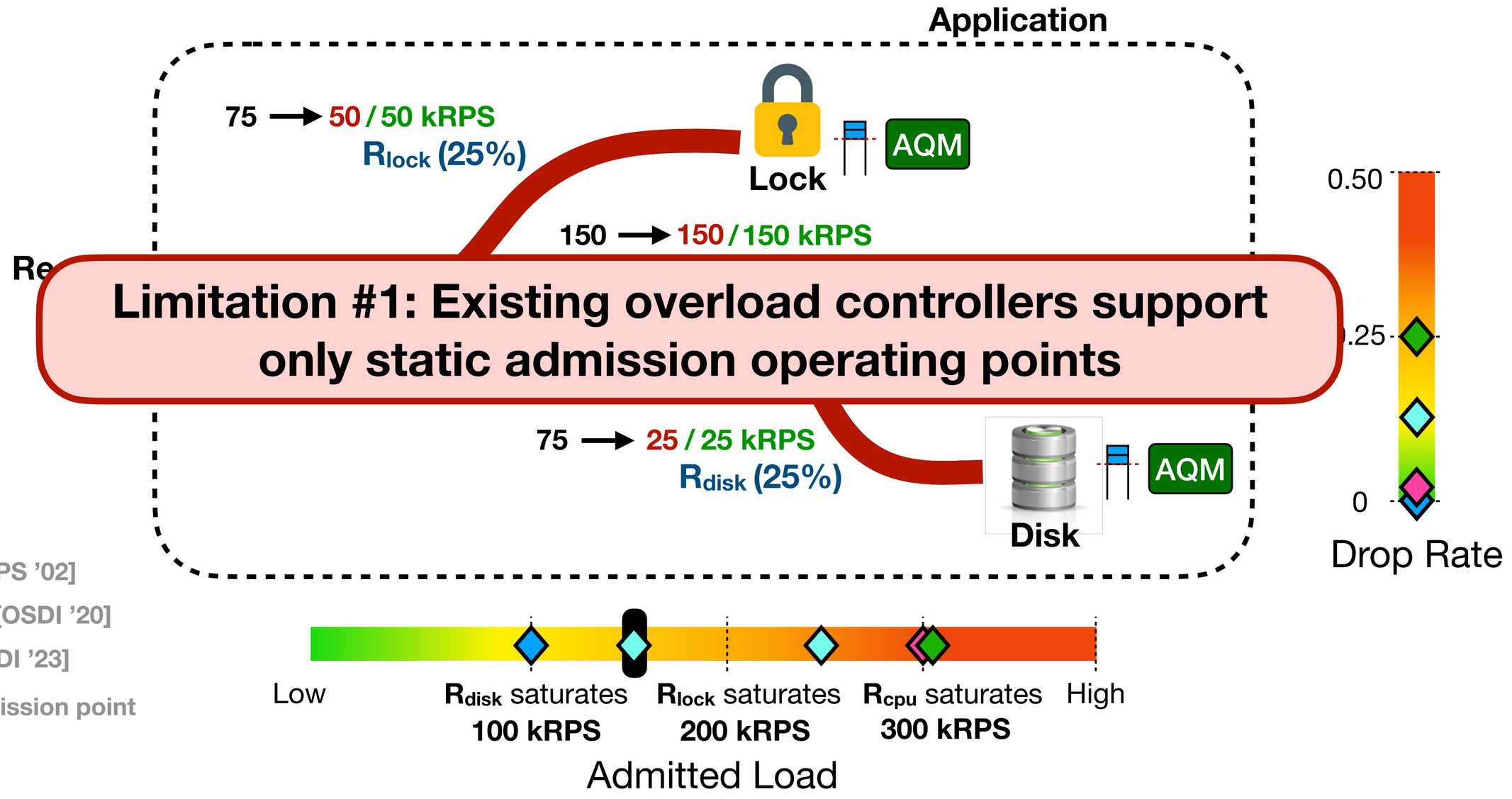




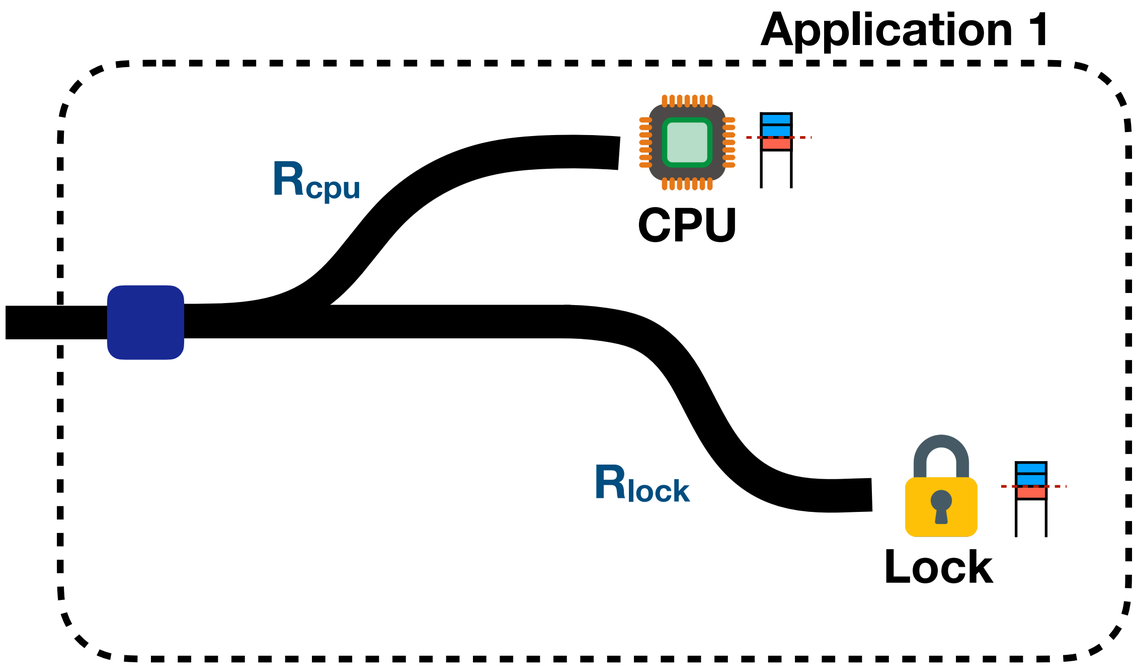
- SEDA [SIGOPS '02]
- Breakwater [OSDI '20]
- Protego [NSDI '23]
- Desired admission point

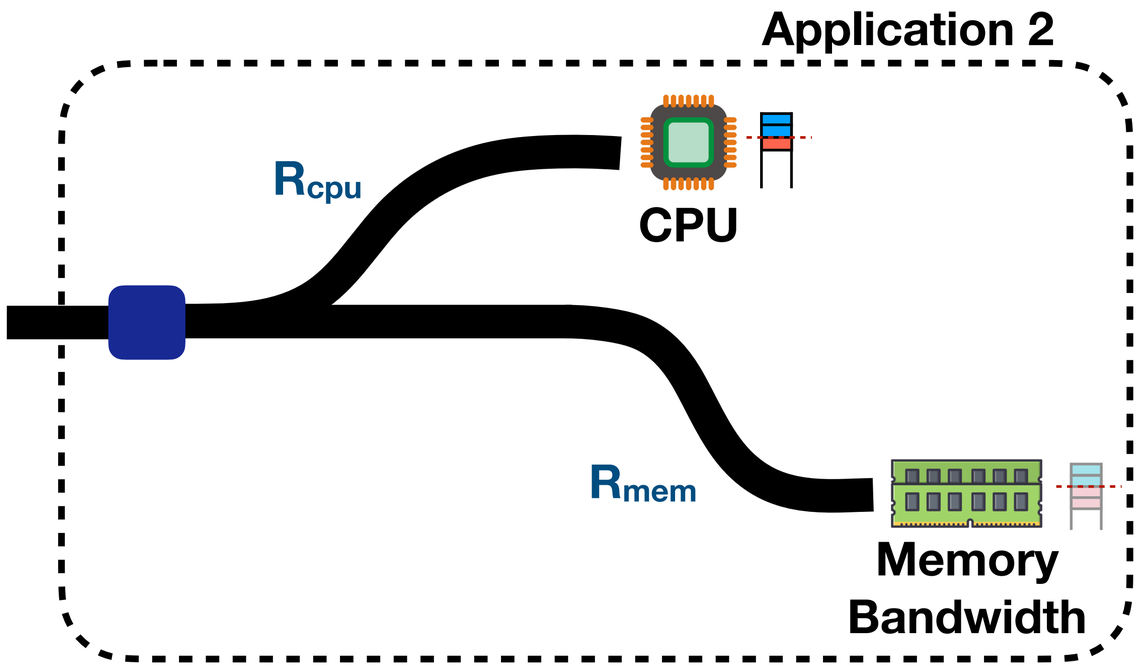


Static Operating Points in the Admission Space

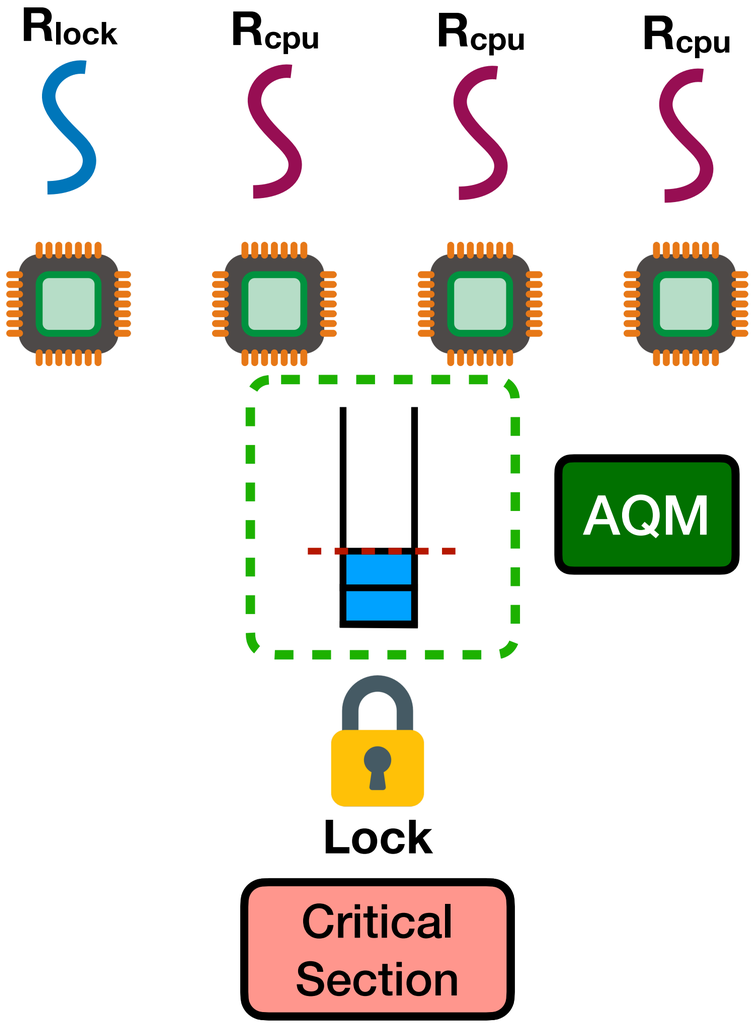


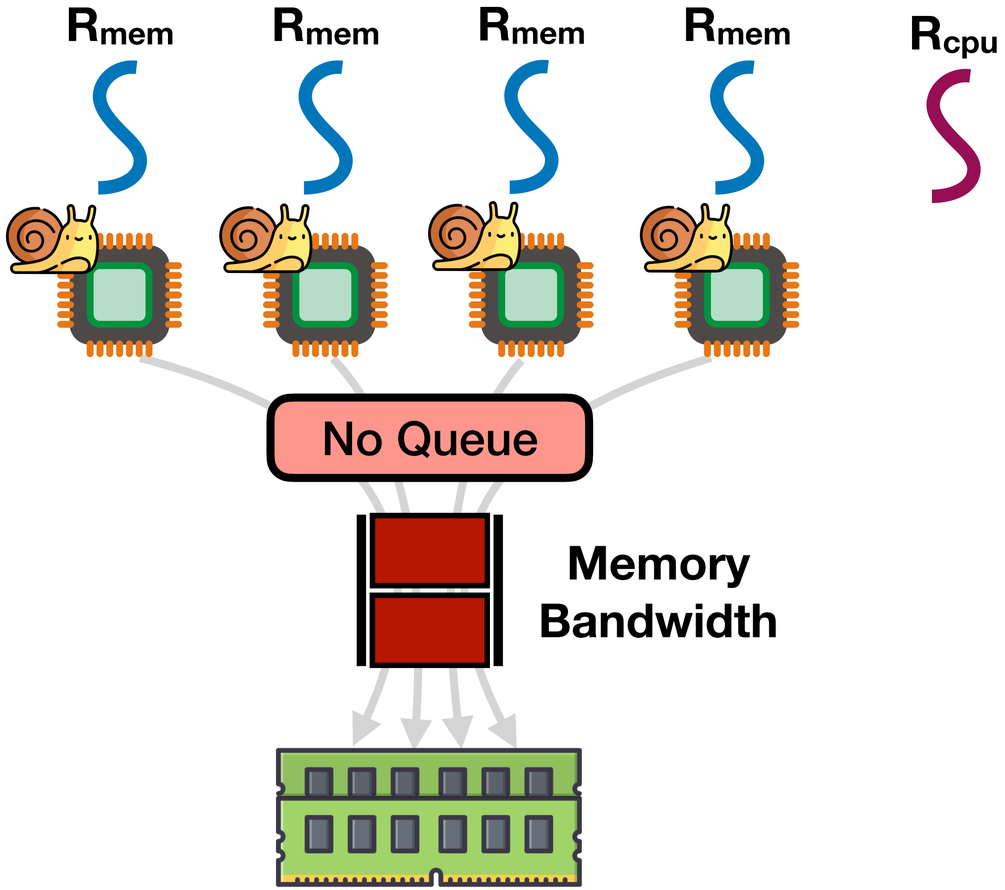
Memory Bandwidth as a Bottleneck



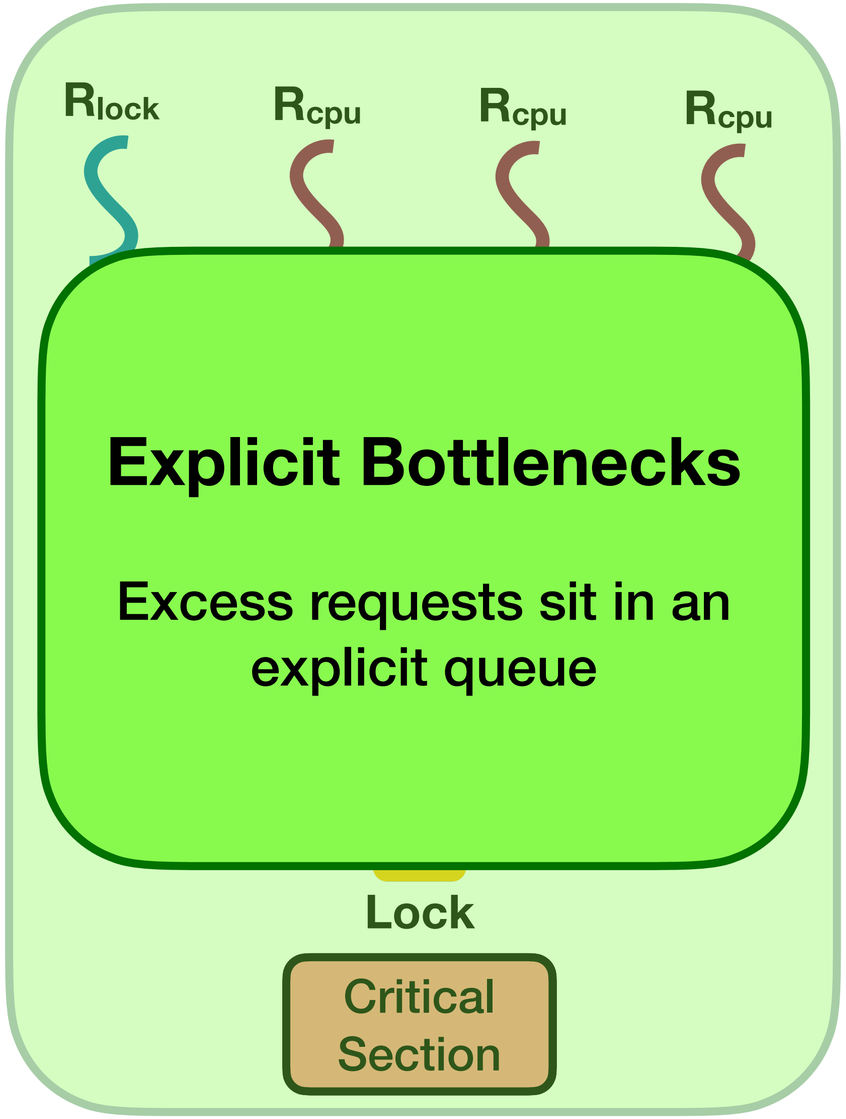


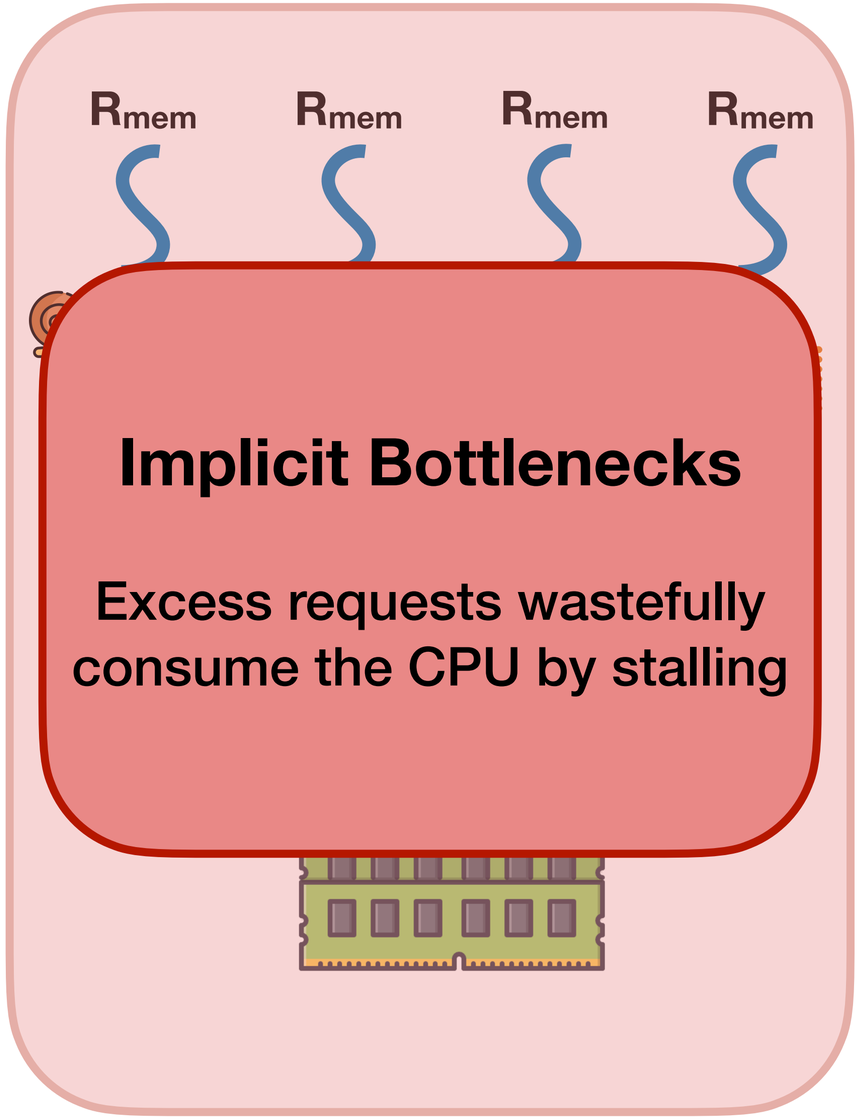
The Curious Case of Memory Bandwidth Congestion



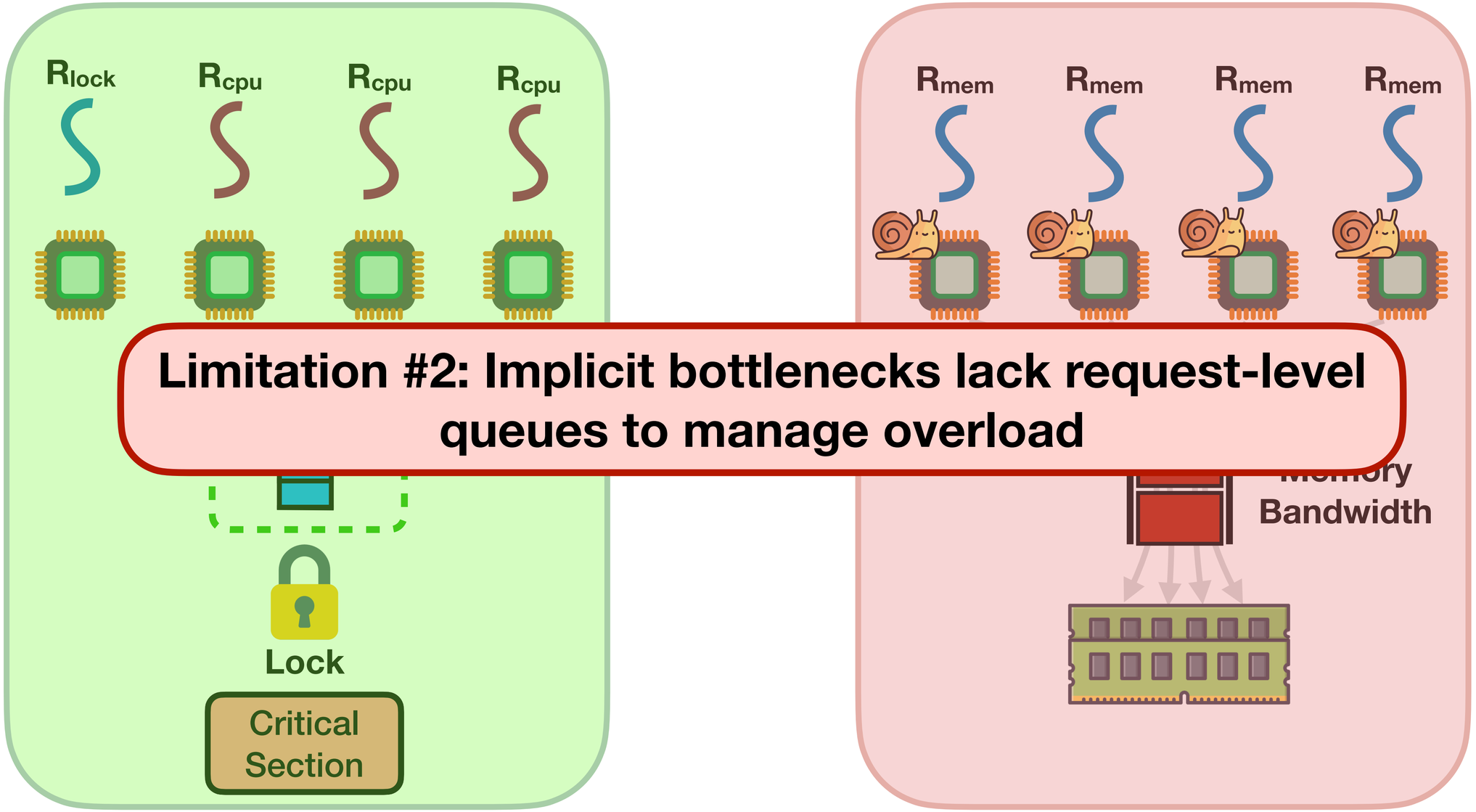


The Curious Case of Memory Bandwidth Congestion





The Curious Case of Memory Bandwidth Congestion

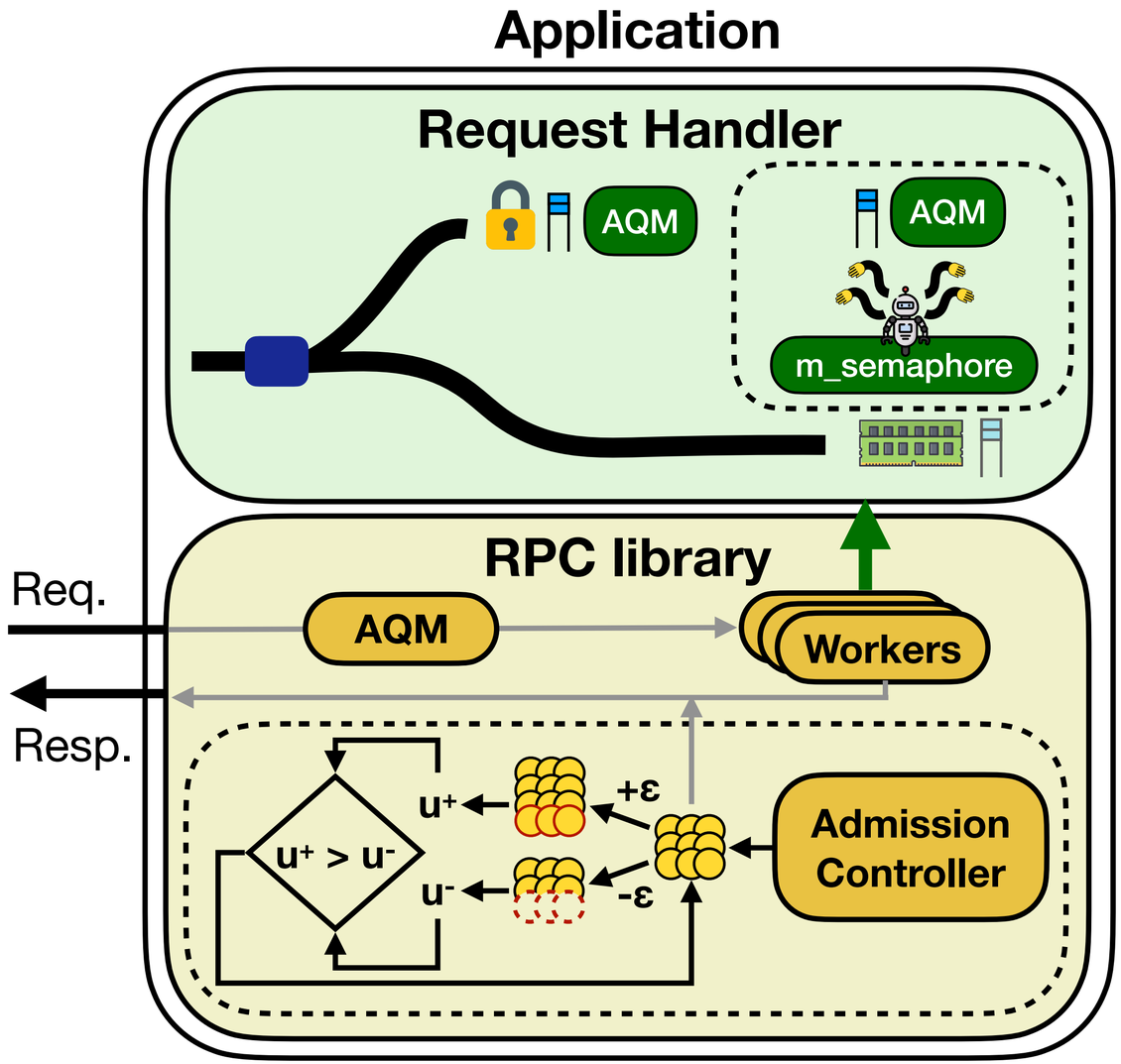


The Design Question

Can an overload controller **admit load to achieve a user-defined objective** while **managing overload at implicit bottlenecks**?

Svalinn

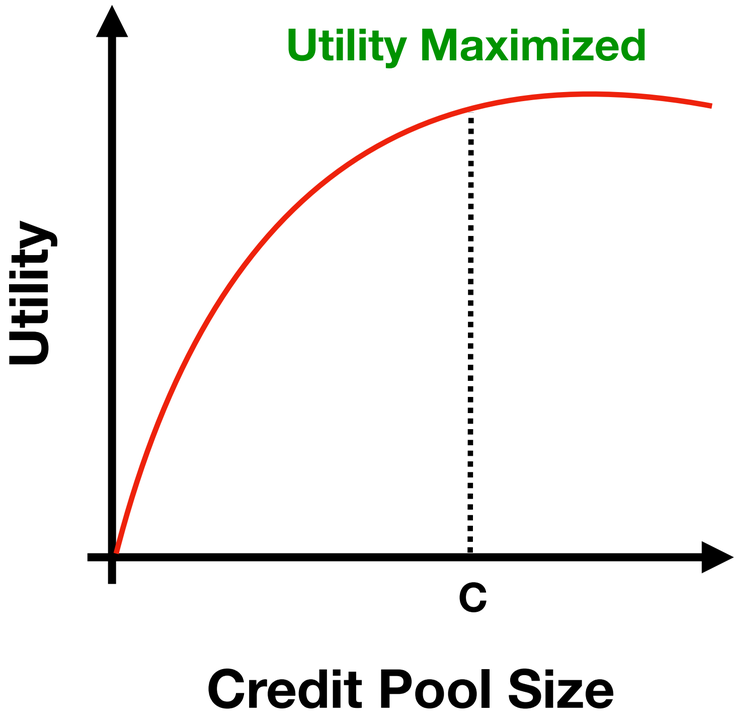
- RPC library for request-response handling
  - **Utility-driven credit-based admission controller** for online learning of the desired admission point
- Controlling latency using per-bottleneck Active Queue Management (AQM)
  - **Multi Arm Bandit-based m_semaphore** for dynamic memory-bandwidth concurrency control
  - Congestion-aware mutexes and condition variables

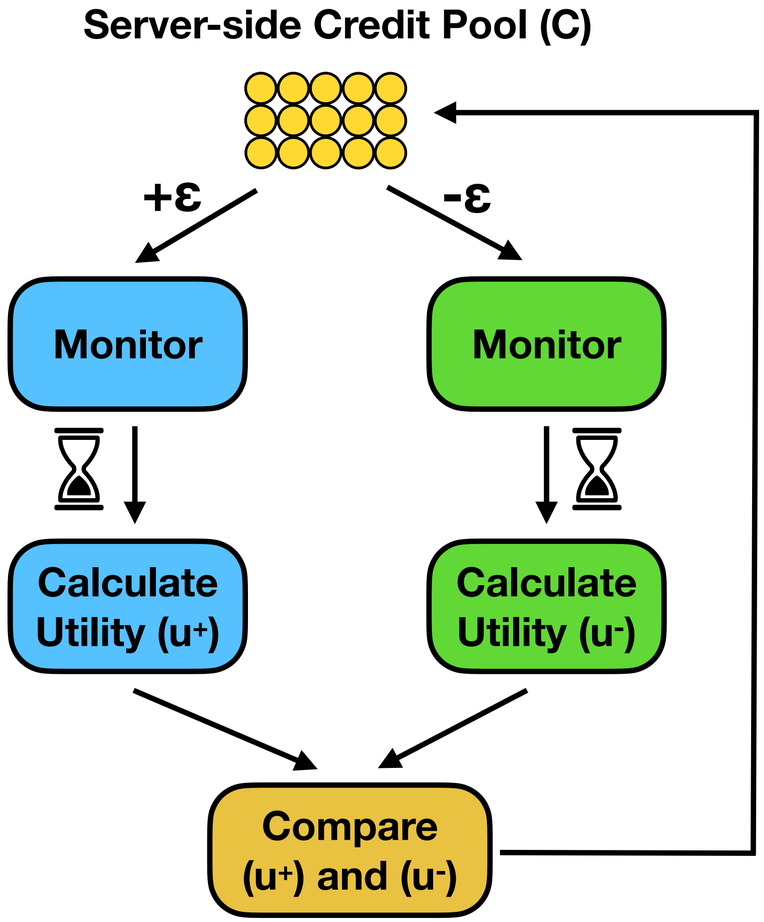


Utility-Driven Admission with Online Learning

Objective: To find a credit pool size that **maximizes** an operator utility

Operator Utility: Maximizing throughput, limiting drop rate, controlling latency, etc.





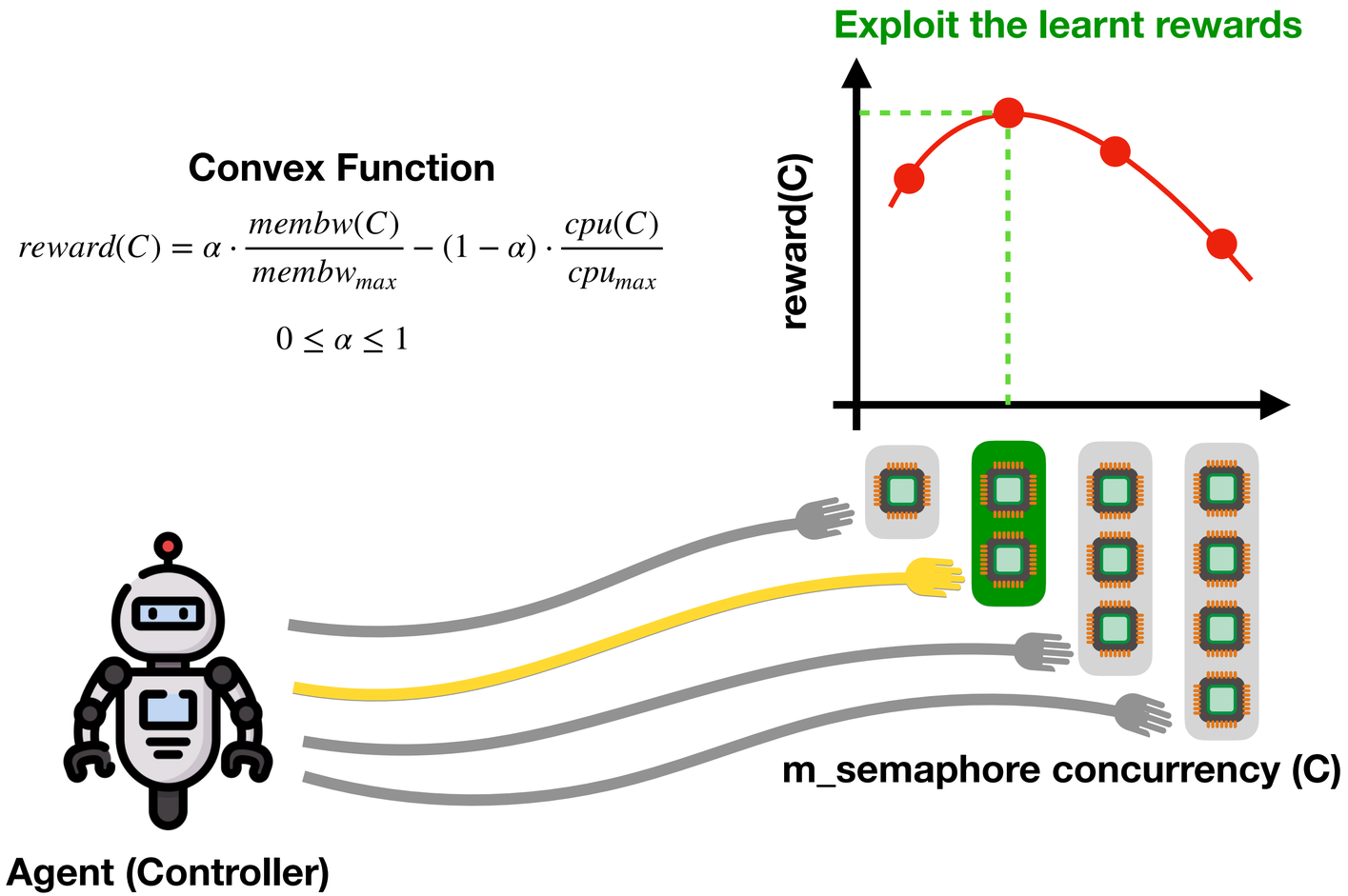
Limiting R_{mem} Concurrency with m_semaphore

Objective: To find a concurrency level that **maximizes** memory bandwidth utilization while **minimizing** CPU usage

Convex Function

$$reward(C) = \alpha \cdot \frac{membw(C)}{membw_{max}} - (1-\alpha) \cdot \frac{cpu(C)}{cpu_{max}}$$

$$0 \leq \alpha \leq 1$$



Exploration-Exploitation Tradeoff managed using ε-greedy Algorithm

Using m_semaphore in the application

```
Response RequestHandler(Request req) {

  Response resp;

  if (m_semaphore.Wait()) {
    ExecuteMembwIntensiveSection(req);
    m_semaphore.Post();
  else {
    req.Drop();
  }

  return resp;
}
```

- Application with **R_{mem}** request type.
- Developers must profile the application to identify memory bandwidth-intensive sections.
- Developers wrap the identified sections with the **m_semaphore's Wait()** and **Post()** calls.
- **Wait()** returns **true** if the AQM policy at the **m_semaphore** queue allows the request to be enqueued.
- **Wait()** returns **false** if the AQM policy predicts that the request may violate the SLO. The developer must drop such requests.

Evaluation Overview

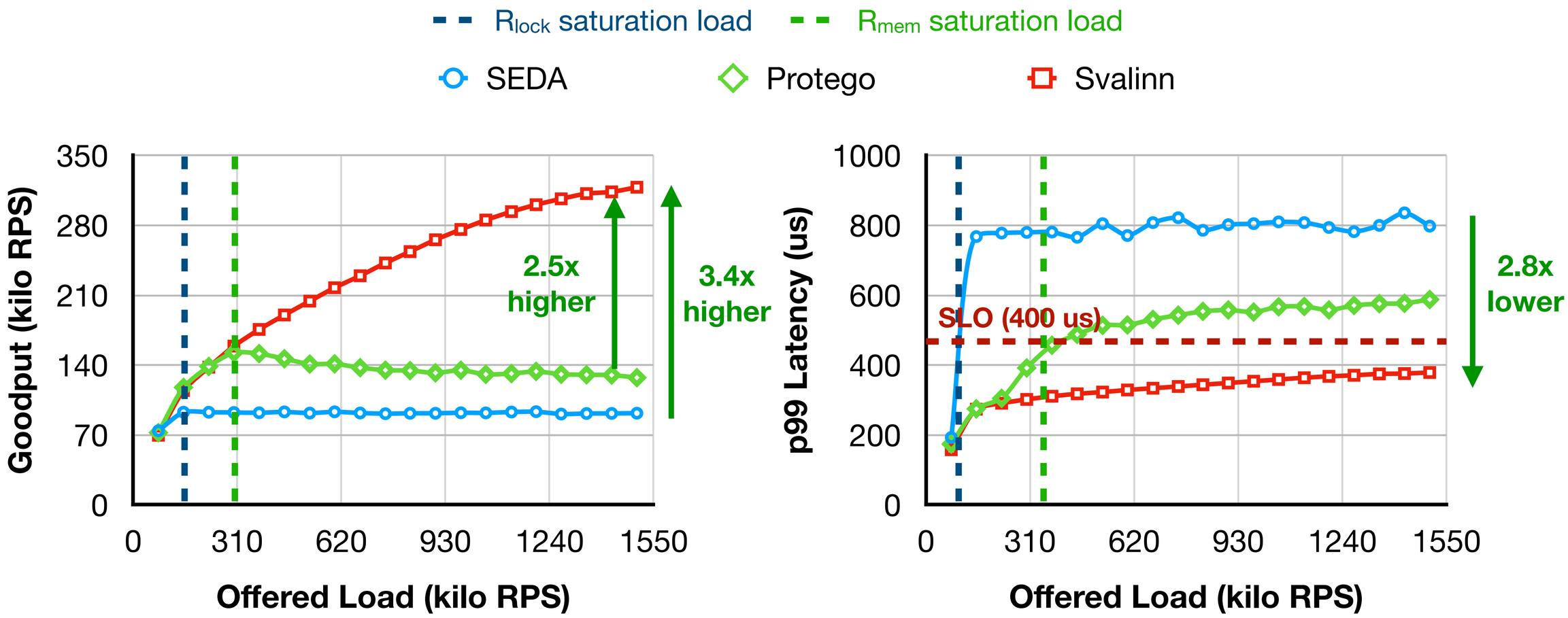
- Evaluation questions
  - How does Svalinn perform under multiple explicit and implicit bottlenecks?
- Testbed
  - 20-core Intel E5-2640v4 @ 2.4 GHz machines
  - 10 client machines generating open-loop Poisson load, and 1 server machine
  - 18 out of 20 cores per machine allocated to the application

Evaluation: Handling Multiple Bottlenecks



- Workload - **Synthetic**
- Request types -
  - 50% of $\mathbf{R_{lock}}$ - Holding a global mutex and performing 5000 square root operations (mean ST 40 us)
  - 25% of $\mathbf{R_{mem}}$ - 25 repetitions of non-temporal stores over a 32 KB in-memory buffer (mean ST 80 us)
  - 25% of $\mathbf{R_{cpu}}$ - 7500 square root operations (mean ST 40 us)
- Svalinn Configuration
  - Admission controller utility - Maximizing Request Throughput $\left(\text{utility}\left(\text{stats}\right) = \frac{\text{stats . out_resp}}{\text{stats . duration}} \right)$
- Schemes to compare
  - **SEDA**
  - **Protego**
- SLO - 400 us

Evaluation: Handling Multiple Bottlenecks



Conclusion

- Modern servers expose **heterogeneous** resource bottlenecks — CPU, locks, storage — including **implicit** ones with no gating API: memory bandwidth, UPI bandwidth, CXL bandwidth.

- Svalinn handles heterogeneity by **decoupling throughput control from per-bottleneck latency control**, and tackles implicit bottlenecks with **m_semaphore** — a new primitive that makes implicit accesses explicit and hence controllable.

bpardeshi3@gatech.edu

