Caladan

Josh Fried

MIT 6.5810

September 19, 2022

Papers

Caladan: Mitigating Interference at Microsecond Timescales [OSDI ‘20]

Josh Fried, Zain Ruan, Amy Ousterhout, Adam Belay

Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads [NSDI ’19]

Amy Ousterhout, Josh Fried, Jonathan Behrens, Adam Belay, Hari Balakrishnan

Need for kernel bypass in datacenters

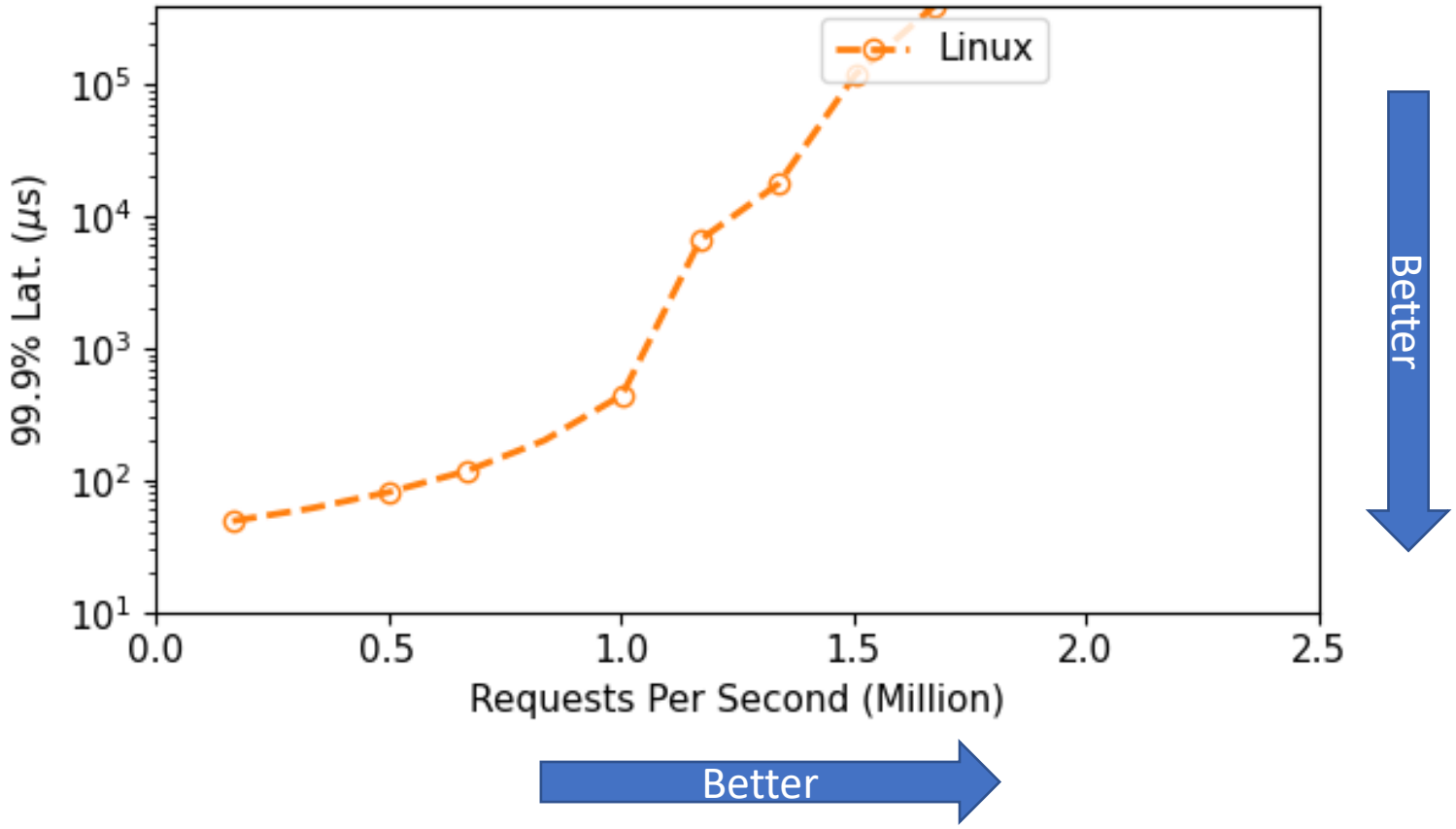
- Using OS for microsecond-scale I/O has high overheads
  - Cost of switching to kernel mode
  - Cost of moving data back and forth
  - Convoluted software paths

- OS thread schedulers + networking stacks cause high tail latency
  - Large quantum (1-4 milliseconds) relative to target SLO (99.9^{th} percentile <= 100us)
  - Interrupt delivery + packet processing threads interrupt application threads
  - Poor packet steering leads to lock contention, cache issues, load imbalance

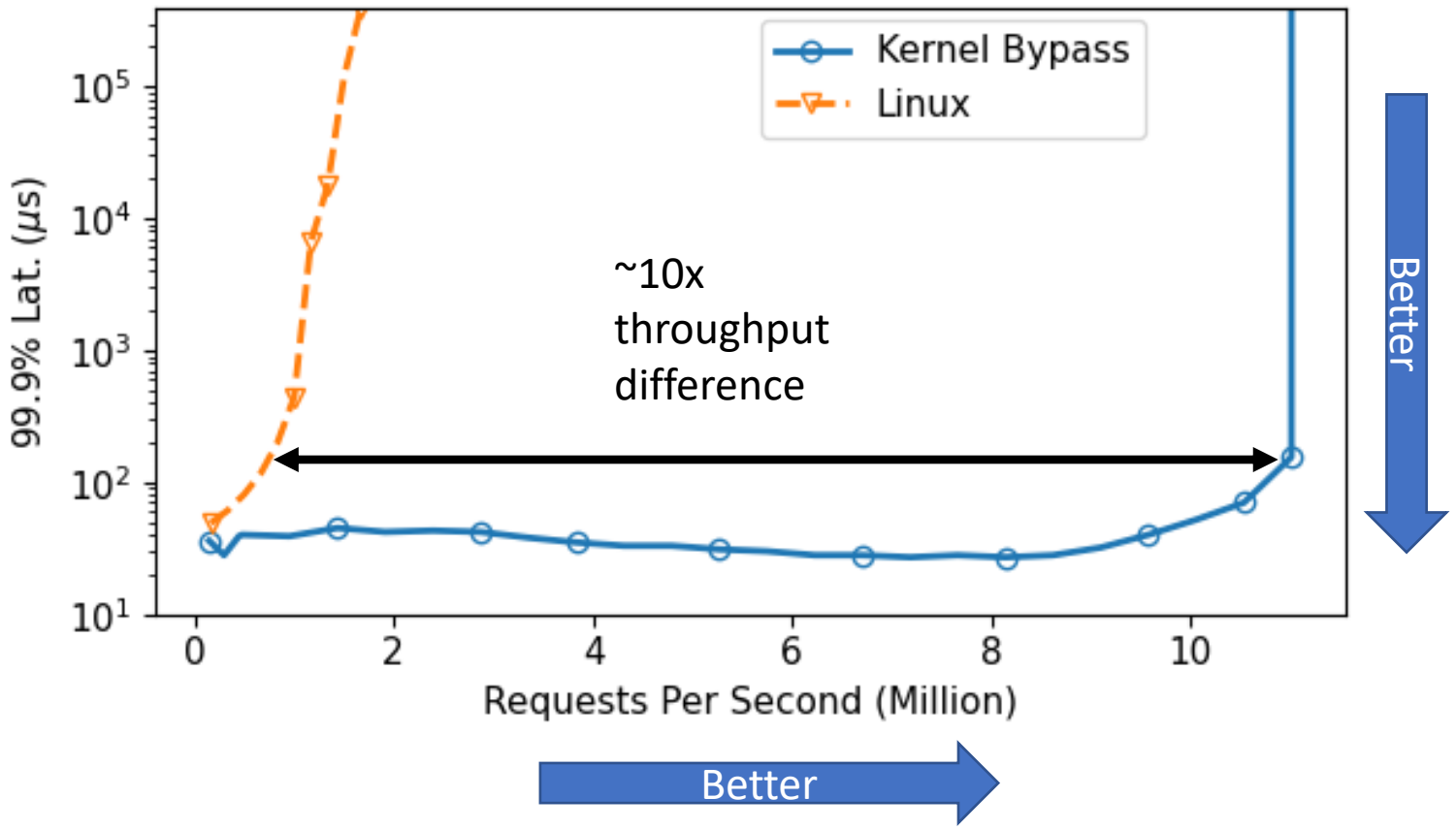
Kernel Bypass

- Control over hardware given directly to application
  - Spin poll hardware queues
    - New packets received, storage commands completed, etc
- Eliminate interaction with OS
  - Dedicated cores
  - Pin memory, use hugepages
  - Avoid syscalls
- Conventional wisdom: avoid shared state across cores
  - Use per-core hardware queues, lockless datastructures, etc
- Examples: Arrakis, eRPC, IX, ZygOS, Shinjuku, Demikernel, [Shenango/Caladan]

Example: Memcached



Example: Memcached



Drawbacks of Kernel Bypass

- Difficult to multiplex resources
  - CPU cores and memory must be pinned
  - I/O devices may not be usable by other applications
  - Load variation makes this particularly challenging

- Programming model
  - Loss of abstractions – working with raw devices
  - No threading or synchronization primitives

CPU-efficient Kernel Bypass

- Performance goal:
  - Provide low latency and high throughput for latency critical applications

- Efficiency goal:
  - Run many applications on the machine to keep it productive
    - Mix of latency critical and best-effort applications
    - Can we keep a machines CPU cores 100% busy with useful work?
    - Avoid overprovisioning resources to latency critical applications

Challenges

- Giving a kernel bypass app the *right* number of CPU cores
  - Too few -> bad performance, high queueing delays
  - Too many -> wasted resources

- How do we change core allocations for kernel bypass apps quickly and efficiently?
  - Slow mechanisms lead to high tail latency

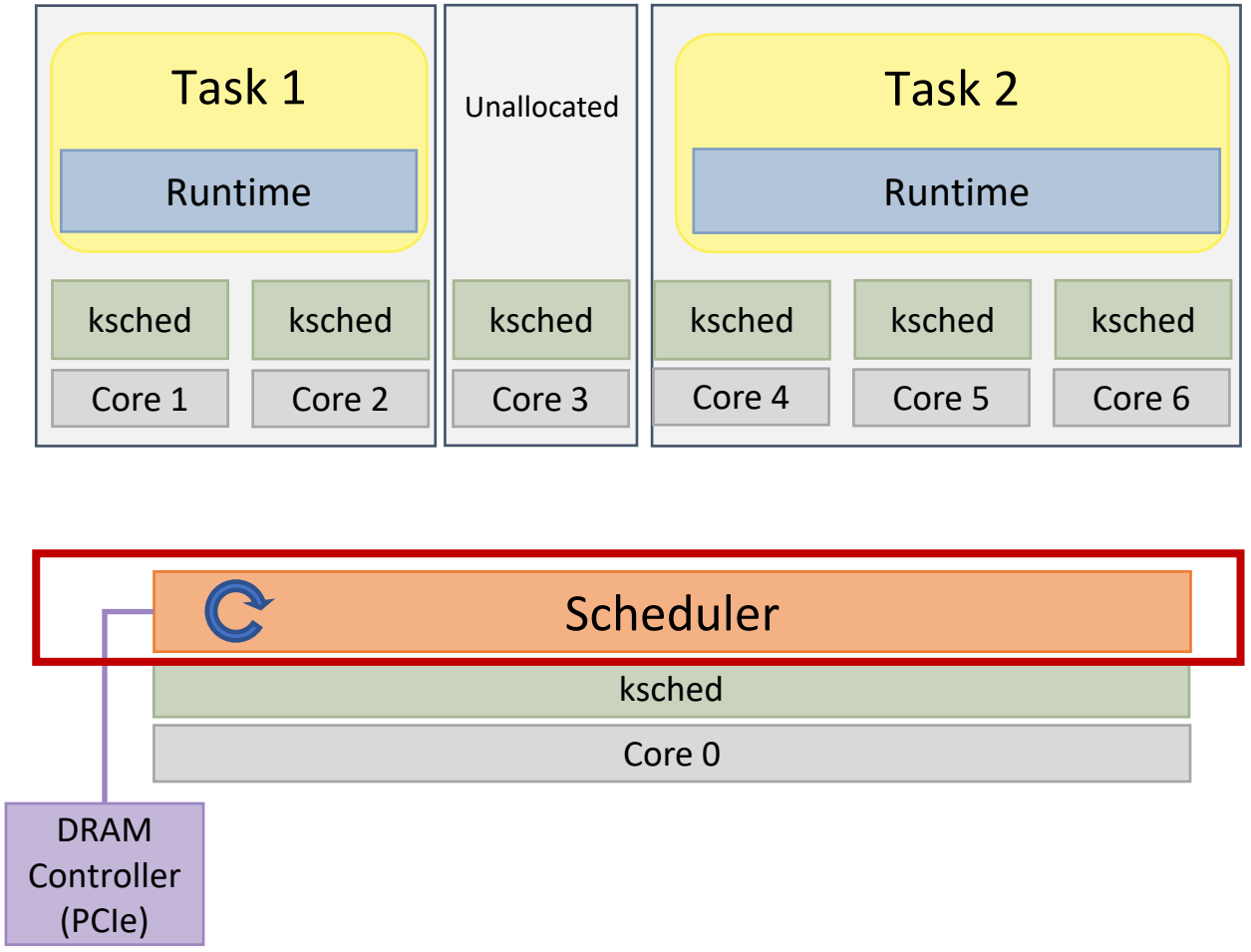
- When packing together many applications on a machine, how do we avoid *interference*?

Caladan: Main Ideas

- Use fine grained core allocations for high CPU efficiency
  - Avoid partitioning resources statically
  - Also use core allocations to control CPU interference
- Provision just enough cores to avoid queueing
- Detect + react to application queueing *before SLOs are violated*
  - 10 microseconds decision interval, can achieve SLOs in 100us range
  - Provision cores when queues build; release them as queueing abates
- Monitor for *causes* and *effects* of CPU interference

Caladan's Components

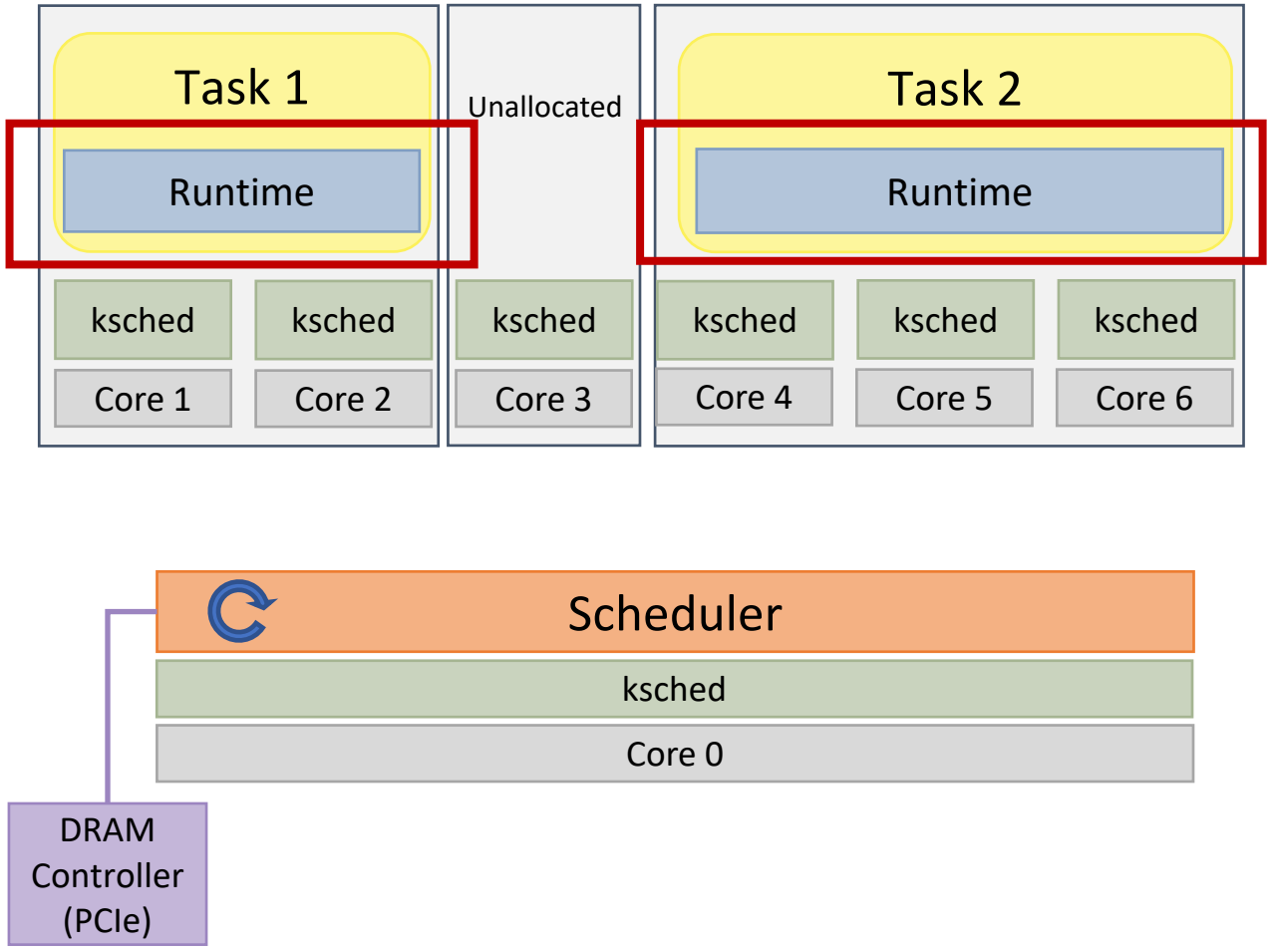
- **Scheduler** core spin polls: monitoring for queueing and signals of interference, assigns tasks to cores



Caladan's Components

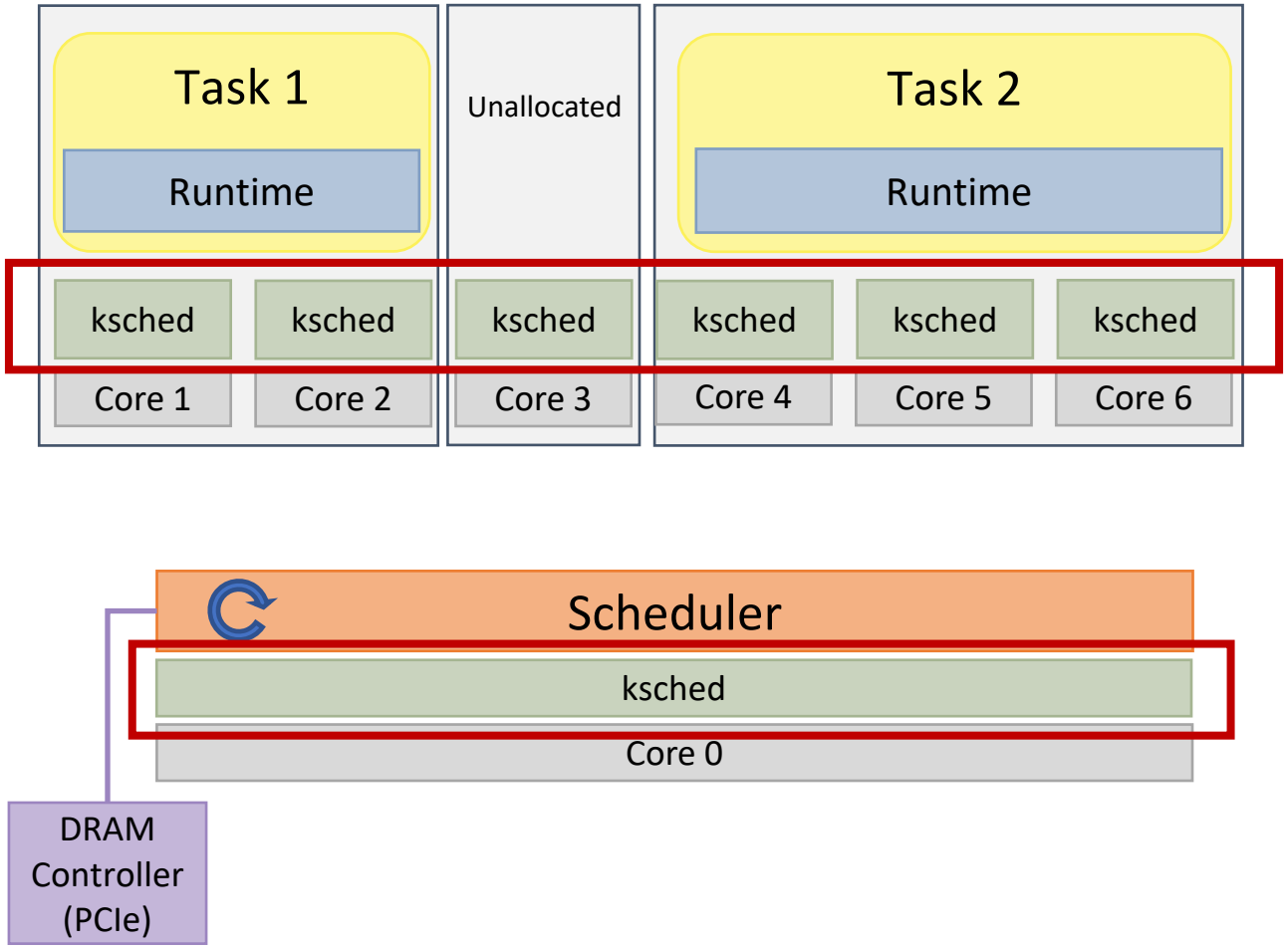
- **Scheduler** core spin polls: monitoring for queueing and signals of interference, assigns tasks to cores

- Tasks link with **runtimes**
  - Provide threading, I/O, etc.
  - Expose signals to scheduler



Caladan's Components

- **Scheduler** core spin polls: monitoring for queueing and signals of interference, assigns tasks to cores
- Tasks link with **runtimes**
  - Provide threading, I/O, etc.
  - Expose signals to scheduler
- **KSCHED** accelerates scheduling and signal gathering

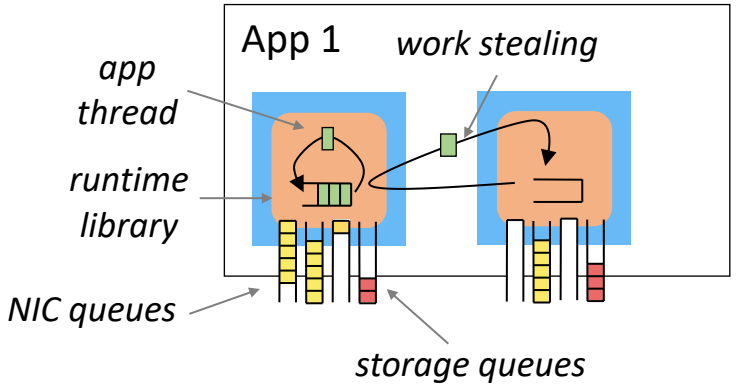


Caladan Runtimes

Lightweight user threading to balance work across active cores

- Per-core thread runqueues + directly-mapped hardware and storage queues
- Thread queues are FIFO, and I/O queues are polled after runqueues are emptied (run-to-completion).
  - I/O completions result in user thread wakeups
- When a core has no work in its runqueue or I/O queues, it *steals* from other cores' queues

FIFO work queues + run-to-completion + work stealing are all important for keeping tail latency low



Runtime Features

- Fast user threading (40ns thread switch time)
- TCP and UDP networking stacks; synchronous API
- Integration with SPDK for fast storage
- High-precision us-scale timers
- Synchronization primitives (mutex, condvar, etc)
- Read-Copy-Update (RCU) for read-mostly shared datastructures

Overall careful design and optimization of the runtime to make these features work well

Detecting Queueing

Each runtime core shares a single cache line (64 bytes) with the scheduler and keeps it updated with:

- current head pointers for runqueue + I/O queues
- timestamp of oldest thread in the queue
- timestamp indicating when current thread began executing
- additional data about the state of the thread

I/O queues instantiated in shared memory to allow scheduler to monitor queueing

Lifetime of a runtime

The runtime creates one linux thread per-core called a *kthread* at startup

- Each kthread *blocks* using a system call until the scheduler core unblocks it
- When woken, a kthread searches for work: attempting to run local user threads, poll/process completions from I/O queues, and steal work from other cores
- If a thread is unable to find *any* work to do, it blocks again and waits for the scheduler to unblock it

Top Level Core Allocation Algorithm

Every 10 microseconds:

```
foreach p in all_procs:
      foreach k in p.kthreads:
            qdelay = 0
            foreach q in [k.runq, k.netq, k.storageq]:
                  # head element in queue is the oldest item
                  qdelay += current_time – q[q.head].enq_time;
            if qdelay >= 10 microseconds:
                  add_core(p)
                  break
```

Policies about granting cores

- Best effort tasks never preempt latency-critical tasks
  - Latency critical tasks can and do preempt latency critical tasks
- Latency critical tasks can preempt each other in certain circumstances
  - For example, if one is using much more than its fair share
- Other constraints imposed when managing CPU interference
  - *Which* cores and *how many* cores are available to application
- Try to select best core given cache locality and HT effects

Preempting a runtime core

Caladan leverages Linux signals to inform runtimes that they are about to be preempted

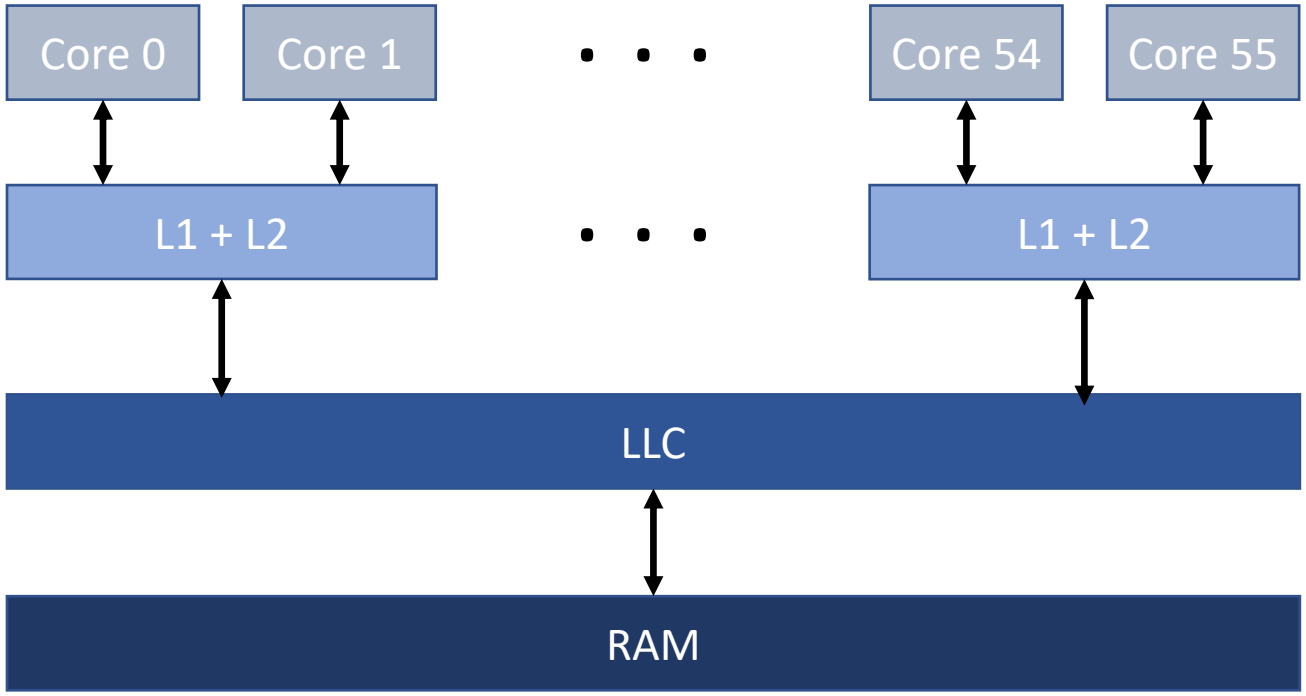
Provide opportunity to cleanly park

- Running uthread placed back in runqueue so it can be stolen by other kthreads
- Defers parking when runtime is in a critical section of code (e.g. for a spinlock)

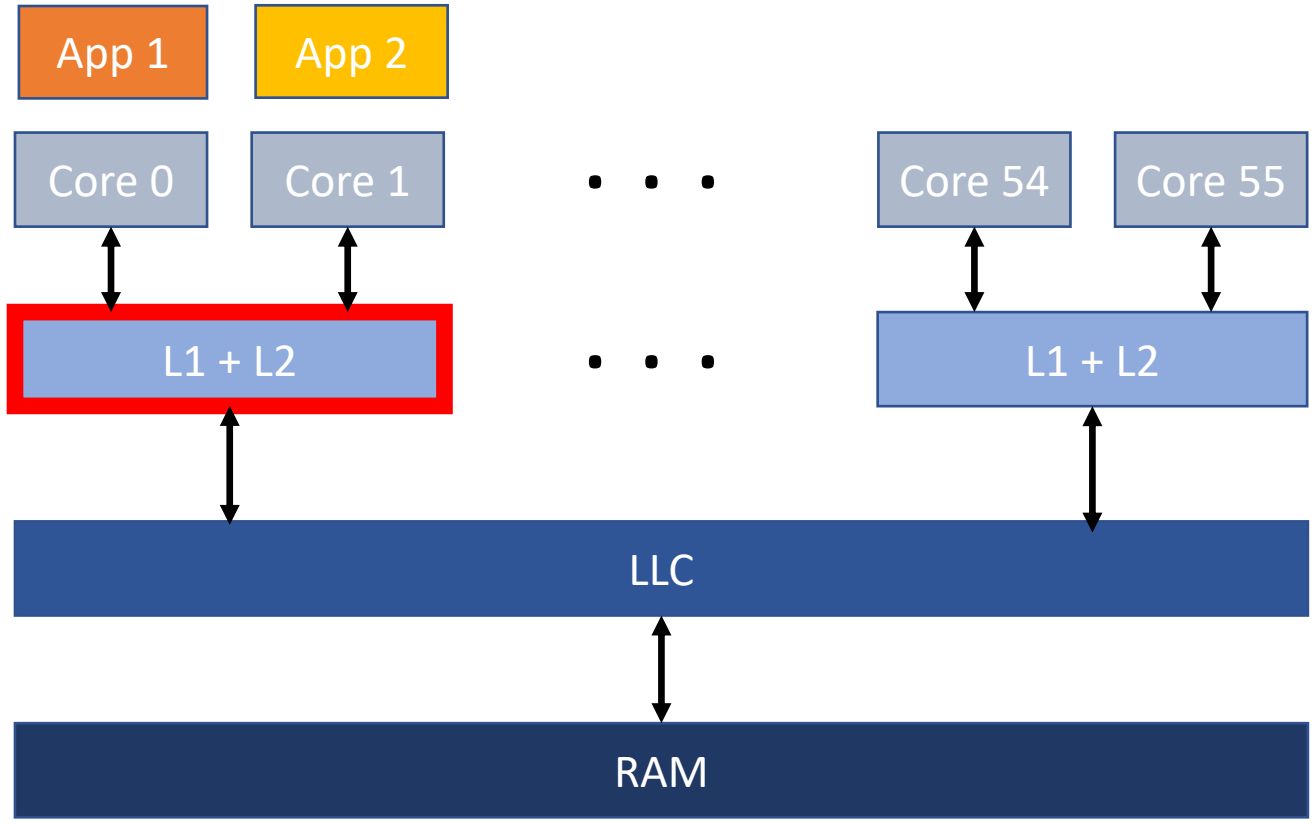
Waking blocked kthreads

- Initial version:
  - Shared eventfd file descriptor between kthread and scheduler
  - kthread issues blocking read() on descriptor, scheduler calls write() to wake it up when it desires
- Latest version:
  - Custom kernel module (*ksched*) provides a better interface for the scheduler core
    - Allows batching of multiple wakeups using multicast IPIs
    - Offloads scheduling work (including signal delivery) to remote cores
    - Shared memory interface for issuing commands, monitoring idle completions, and more

Interference in Shared CPUs

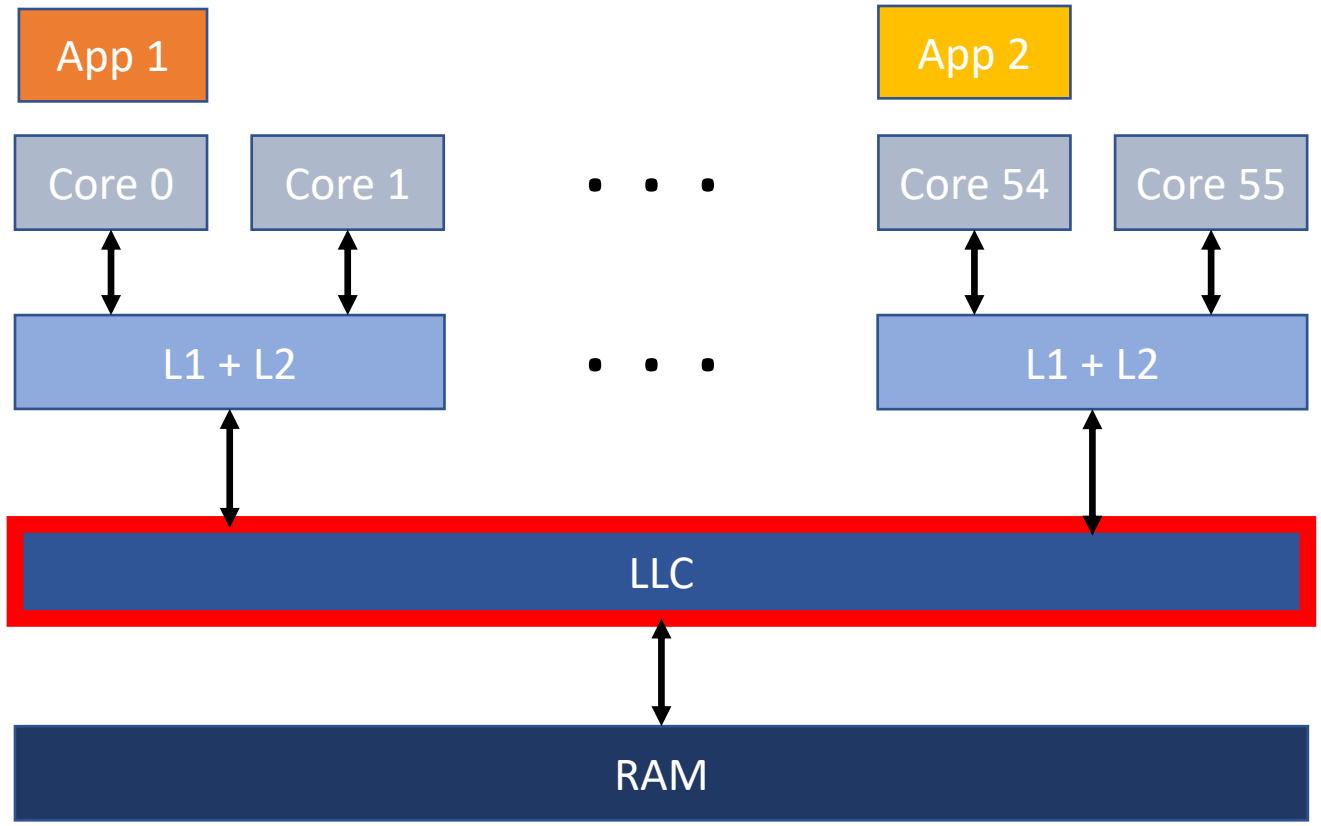


Interference in Shared CPUs



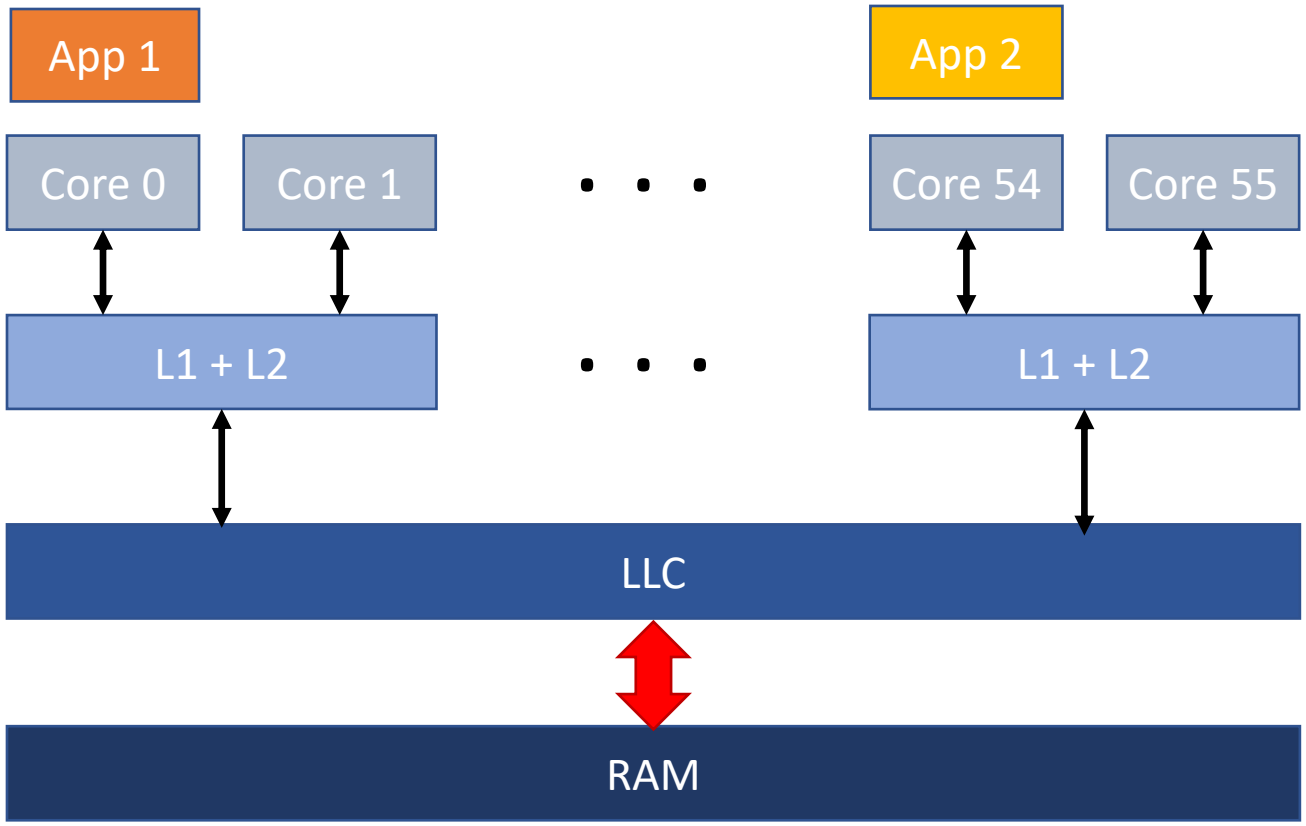
Hyperthreading Interference

Interference in Shared CPUs



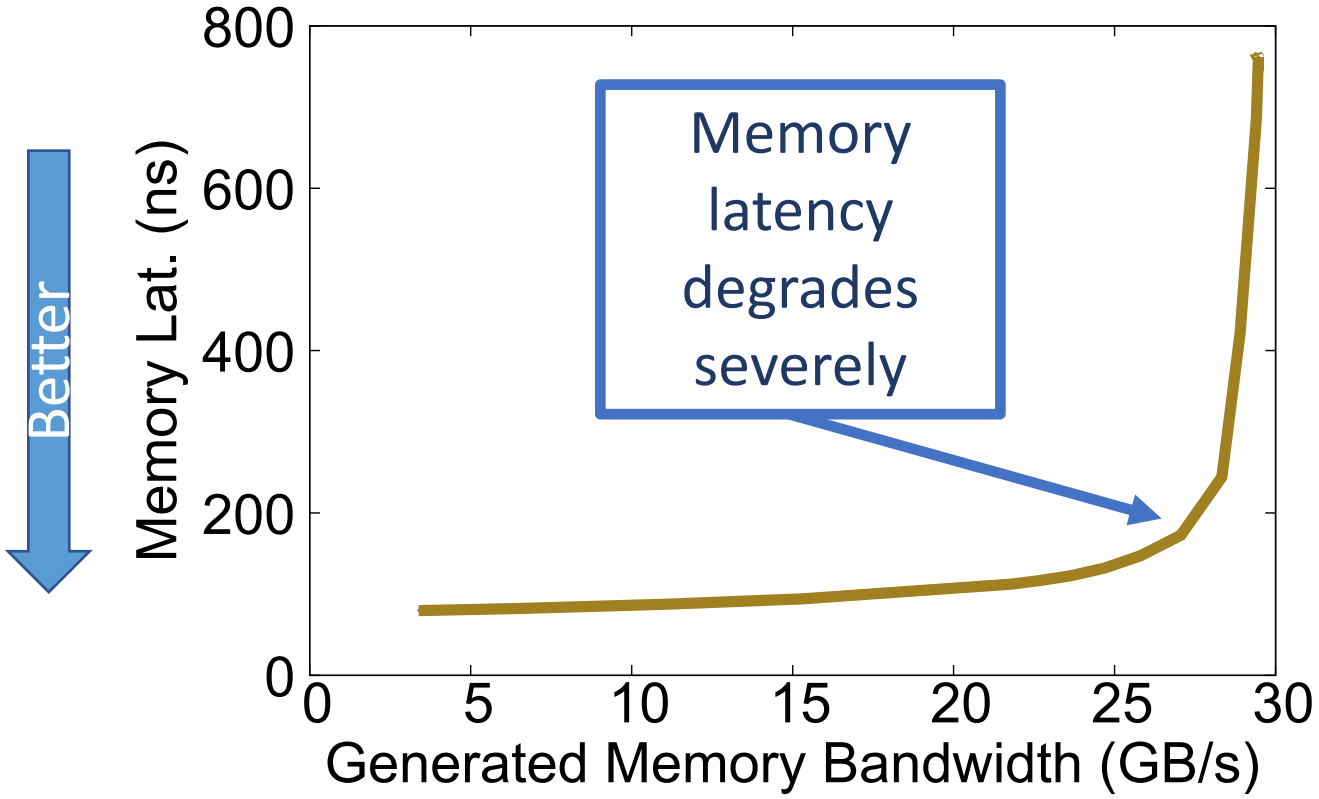
LLC Interference

Interference in Shared CPUs



Memory Bandwidth Interference

Managing Memory Bandwidth Interference



- Policy: keep total bandwidth below target (~80%)
- Detecting Bandwidth Usage:
  - DRAM controller counters (checked every 10us)
  - Per-core LLC miss counters for task attribution
- Action: Throttle core count for high bandwidth tasks

Managing Hyperthread Interference

- Pair different tasks on hyperthread sibling cores
  - Many other systems disable hyperthreads all together, can be up to a 30% loss in machine throughput
- Policy: Allow an LC *request* to execute on the same physical core as another task for a fixed amount of time (THRESH_HT)
  - Slowdown due to HT interference is bounded by THRESH_HT
  - Positive effect on tail latency even when interference is not severe
- Action: Prevent any task from executing on the hyperthread twin core when an LC request has exceeded THRESH_HT in execution time
- Generalized strategy from Elfen Scheduling [ATC ‘16]
  - Caladan improves utilization by allowing arbitrary task pairings

Managing LLC Interference

LLC interference is the least extreme of the three types

No active policy to manage it

Instead, it is sufficient to rely on the top-level core allocation algorithm to make up for any lost processing capacity

Implementation

Scheduler

- Optimized to run the full control loop every 10 µs
- 3500 LOC
- Currently supports Intel CPUs

KSCHED – Linux kernel module

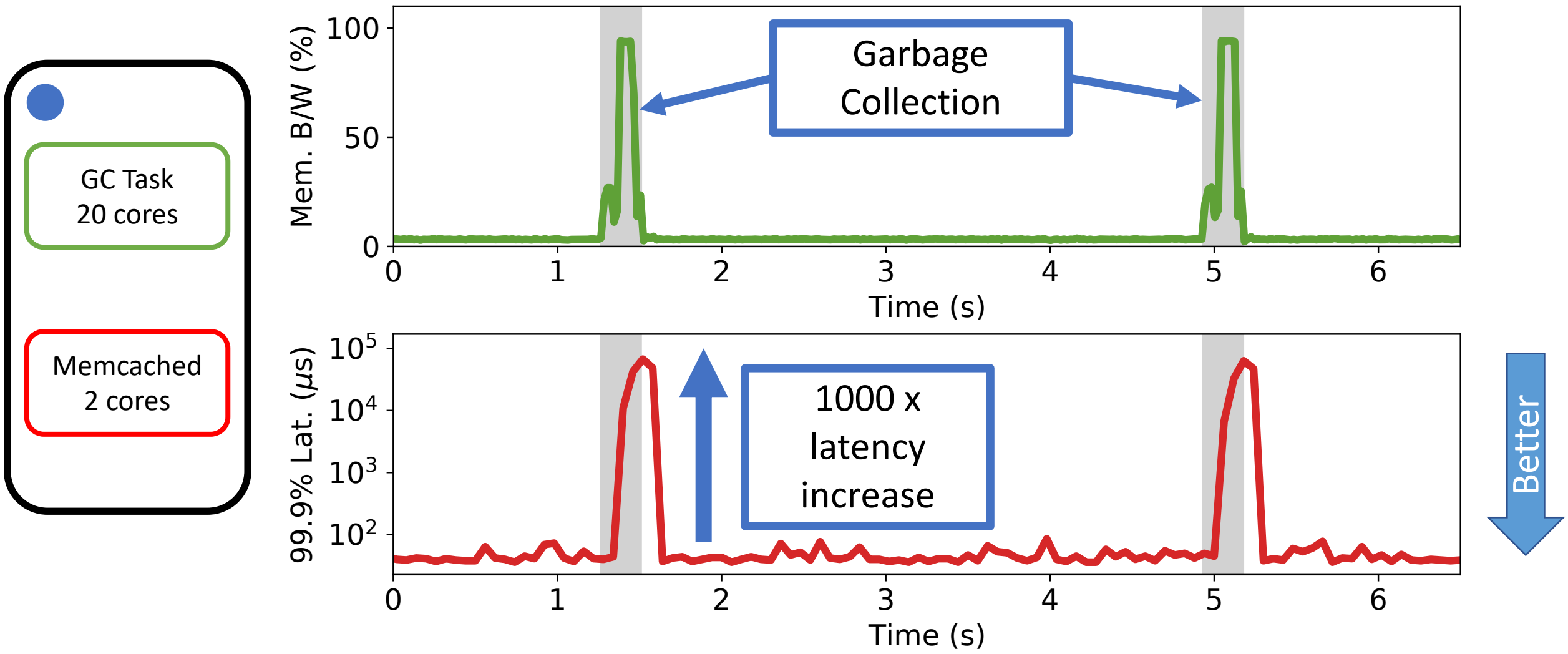
- Leverages hardware multicast IPIs
- 530 LOC

Runtime

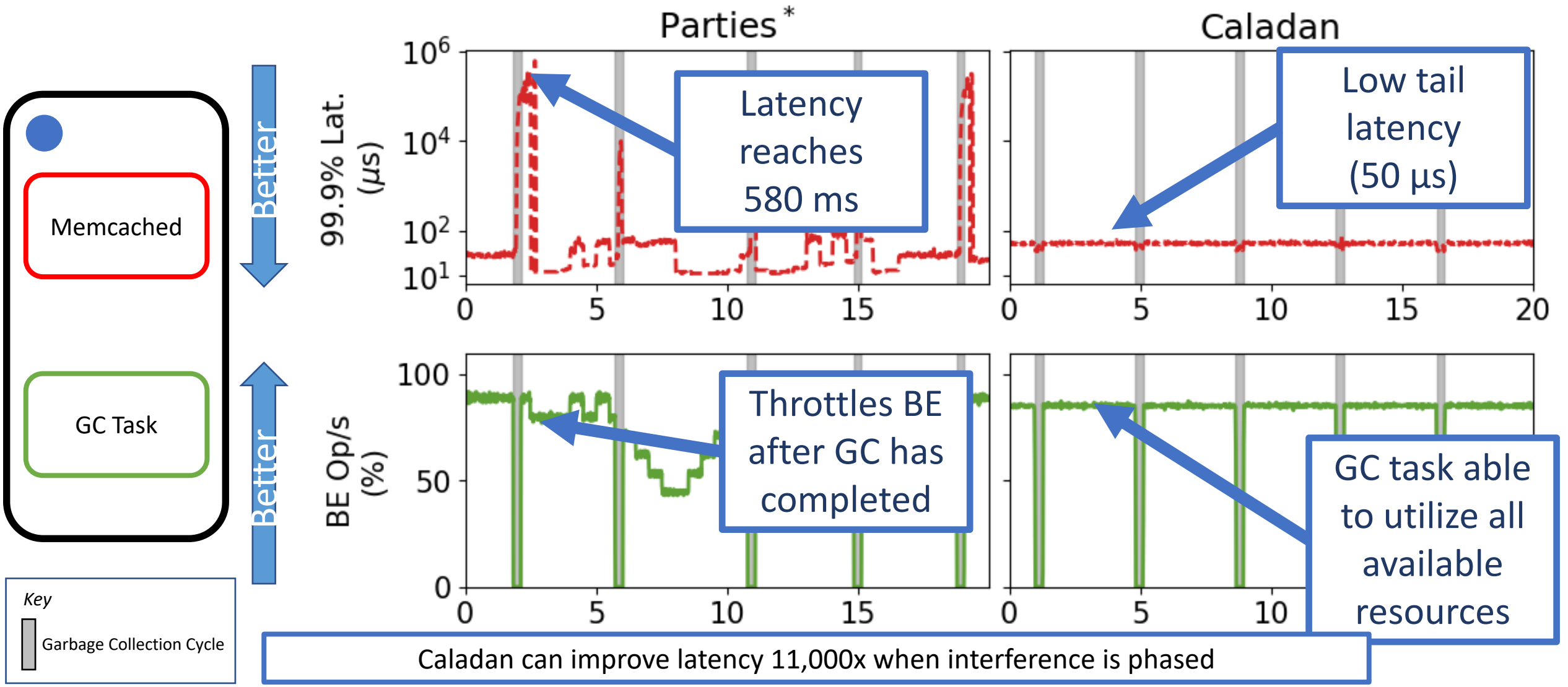
- ~10,000 LOC
- Custom mlx5 driver for Mellanox ConnectX-5+ NICs
  - Fast flow steering for fast core reallocation
  - Exposes queueing signals to scheduler

Available at https://github.com/shenango/caladan

Interference Example



Memcached and GC



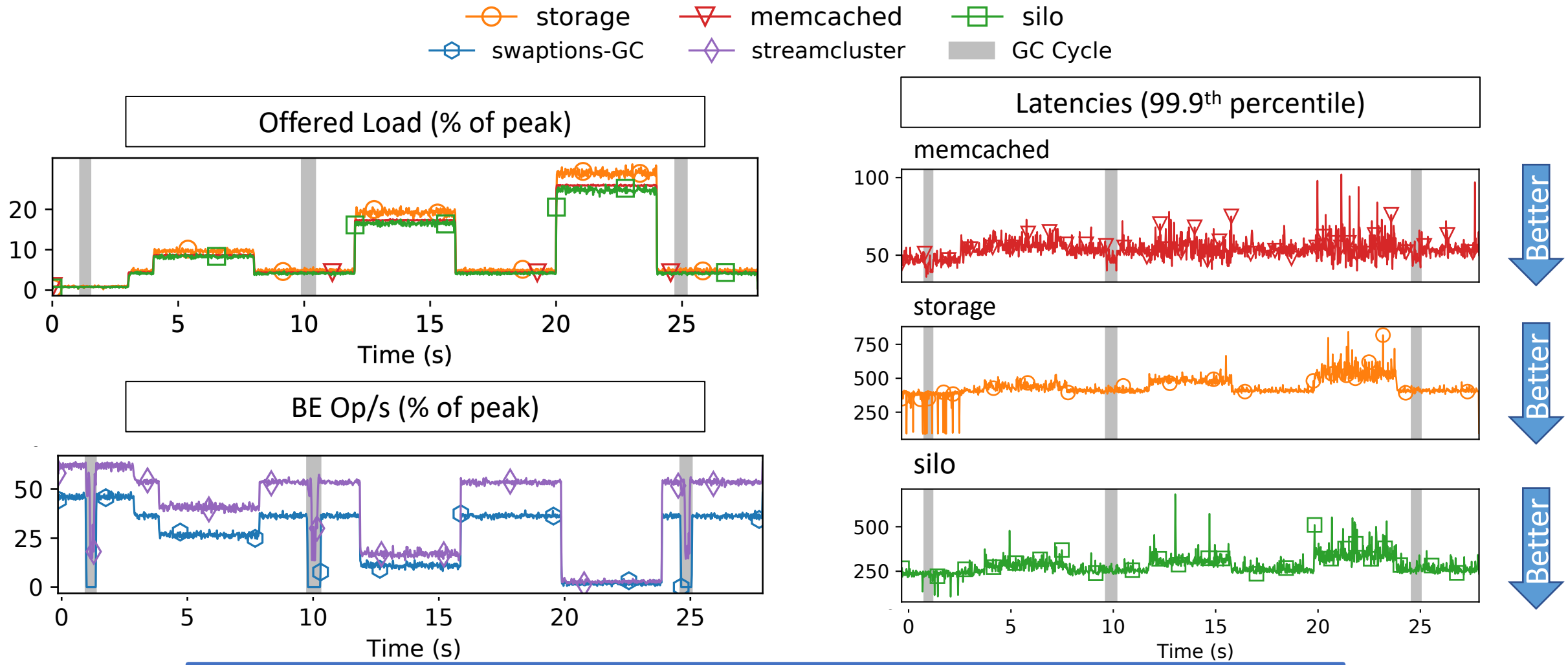
Colocating Many Tasks

- 3 Latency-Critical Tasks
  - Memcached
  - Flash storage service
  - Silo
- 2 Best-Effort Tasks
  - Swaptions (GC Task)
  - Streamcluster

30 seconds, variable load and interference

Colocating Many Tasks

Core allocations occur up to 230,000 times per second



Caladan maintains low tail latency for all 3 LC tasks under varying load and interference

Requirements for Applications

Applications must link with the runtime

- Export signals, balance work across active cores
- Realistic programming model
  - Partial compatibility layer for some systems libraries

LC applications must expose internal parallelism to runtime

- Example: Memcached modified to spawn a thread per-connection

- Allows scheduler to observe delays
- Allows scheduler to mitigate delays with additional cores

No required changes for BE tasks

Overload Control

(feat. Breakwater)

Inho Cho

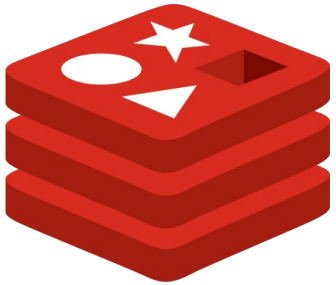
Latency is an important metric in DC

- It directly impacts a user experience
  - E.g. How long does it take to fetch a website?
  - E.g. It determines how fast you decide whether to buy/sell stocks
  - Especially for interactive applications like Cloud gaming

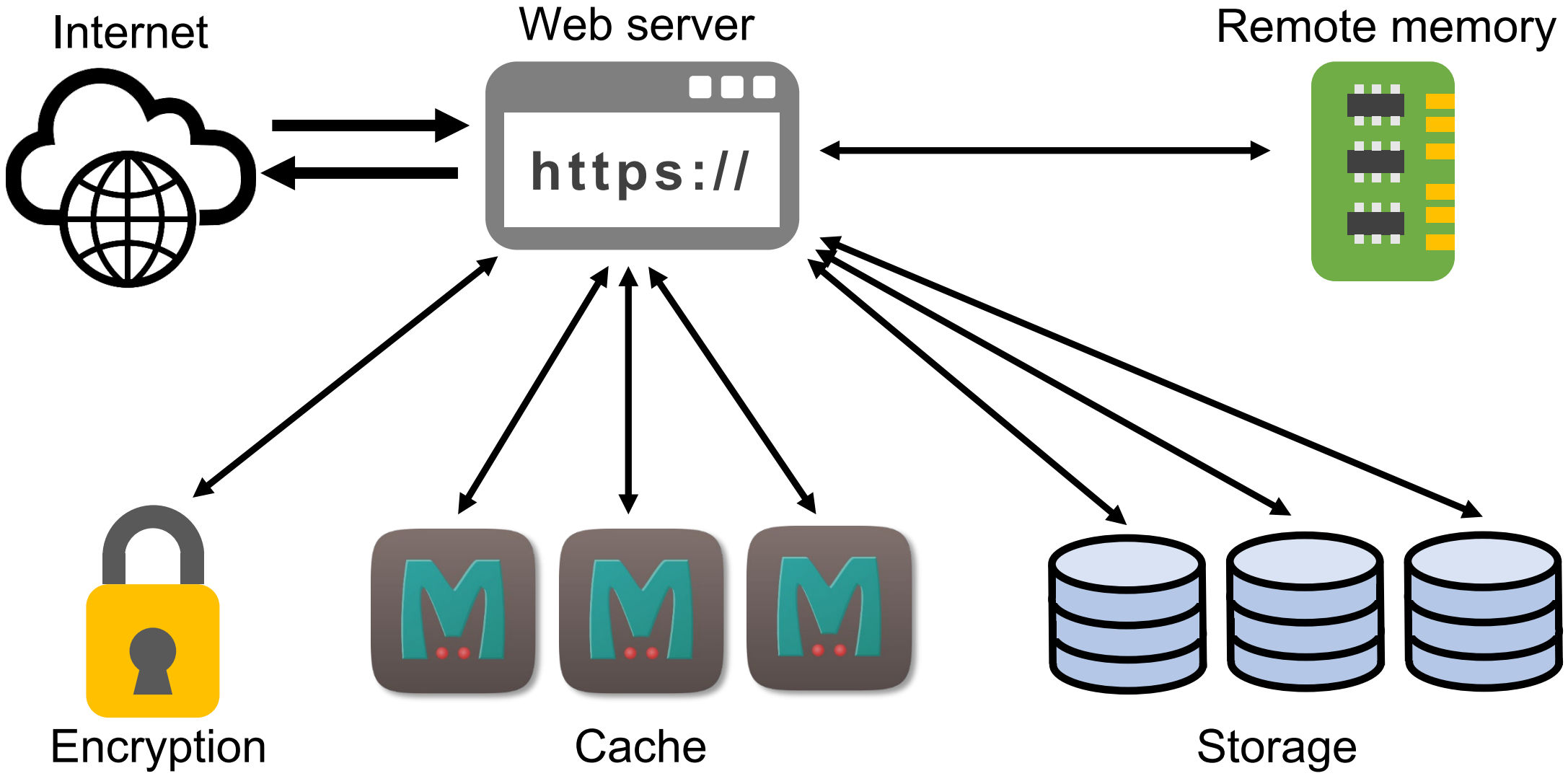
- SLO (Service Level Objective) is often defined by latency

Many efforts to reduce the latency

1. Fast Network: Network latency (~ 5 us)
2. Fast Storage: M.2 NVME SSD (~ 20 us)
3. In-memory operations: Memcached, Reddis, Ignite

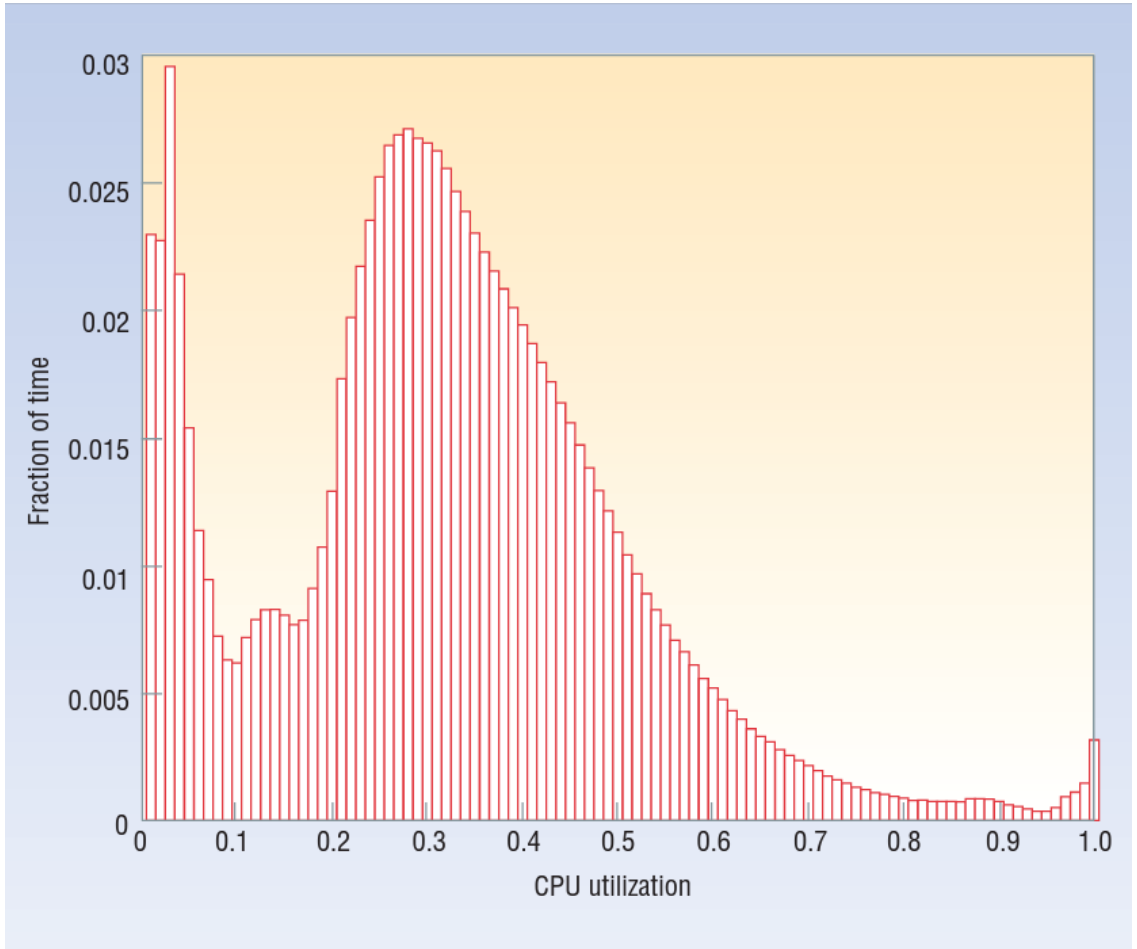




Trend: High Fan-out



Traditional Solution

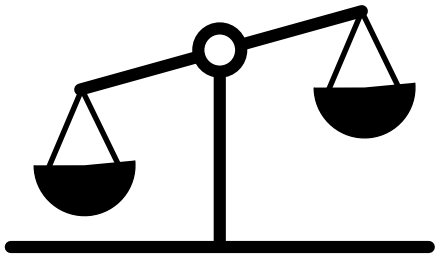
Keep CPU utilization low to ensure low latency by **over-provisioning** the resources



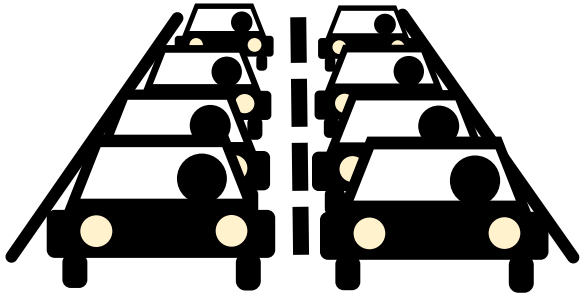
Barroso, L.A. and Hölzle, U., 2007. The case for energy-proportional computing. Computer, 40(12), pp.33-37.

Causes of Server Overload

Load Imbalance

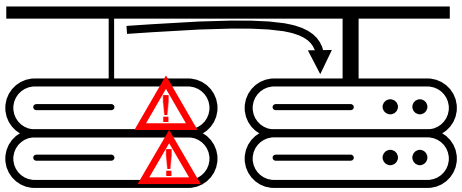


Unexpected user traffic

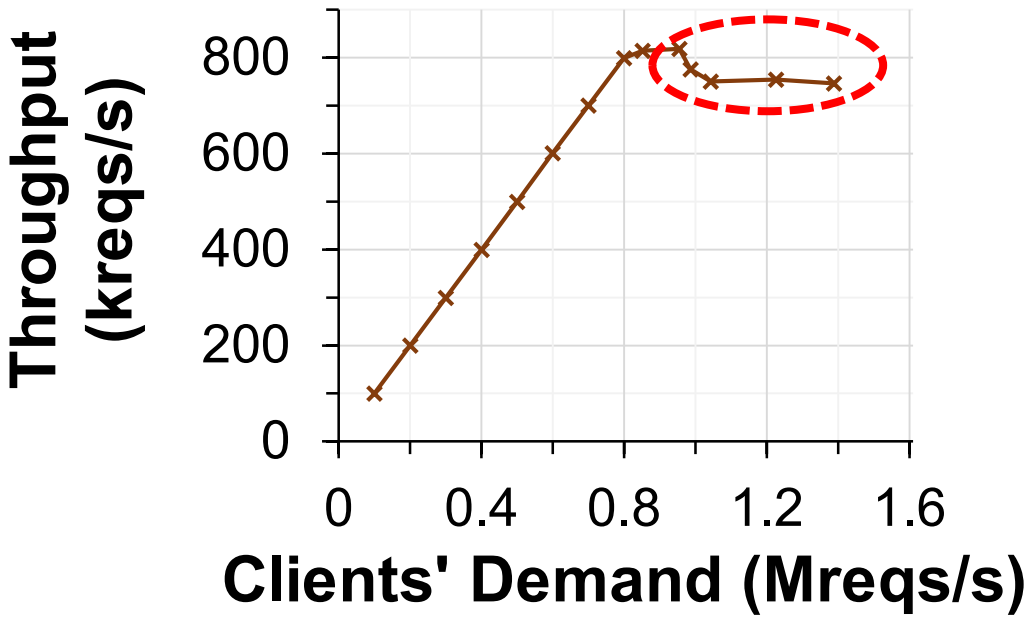


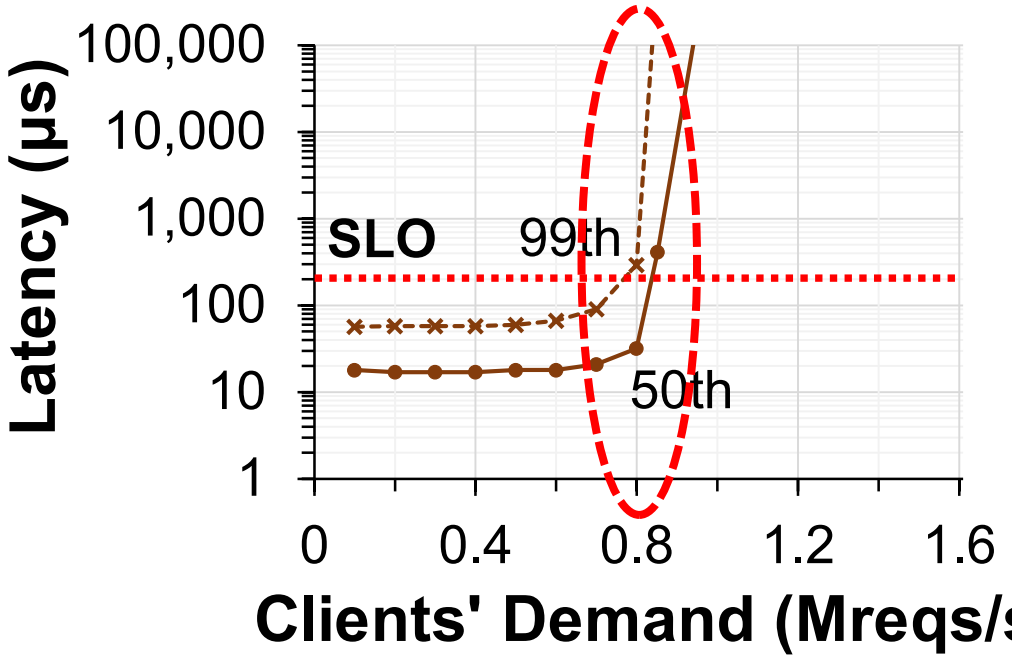
Packet bursts

Redirected traffic due to failure



What happens when the server is overloaded?

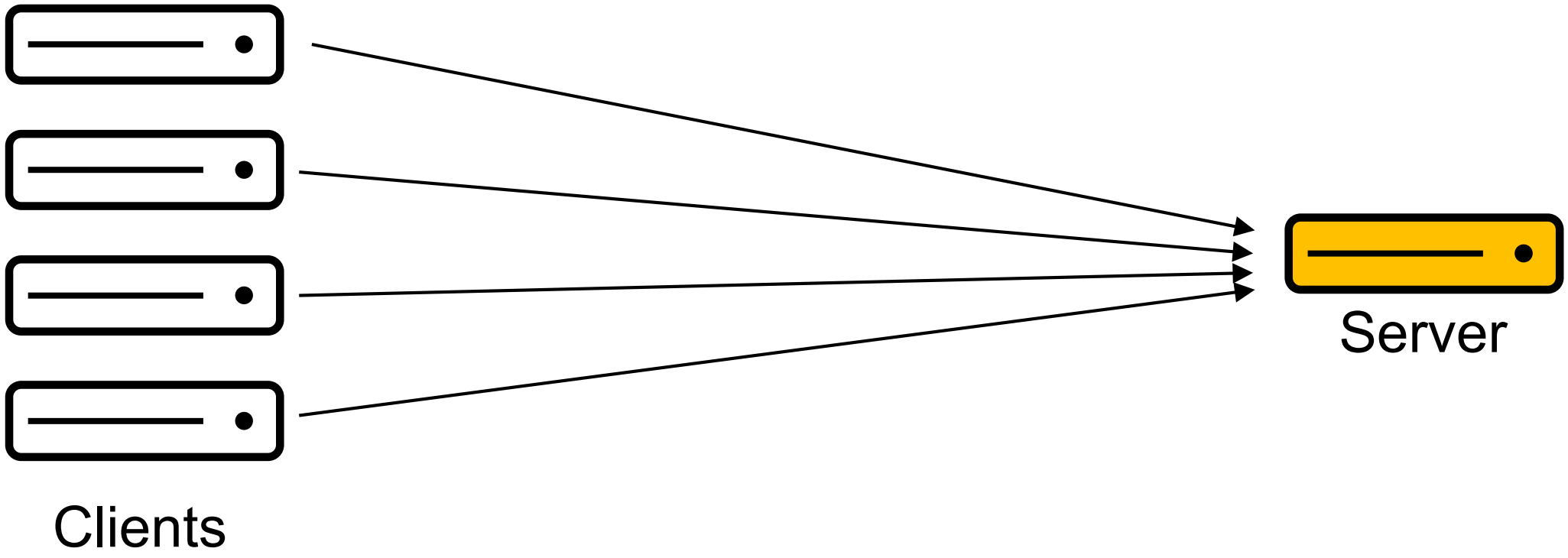




Without overload control, server overload makes almost all requests violate its SLO.

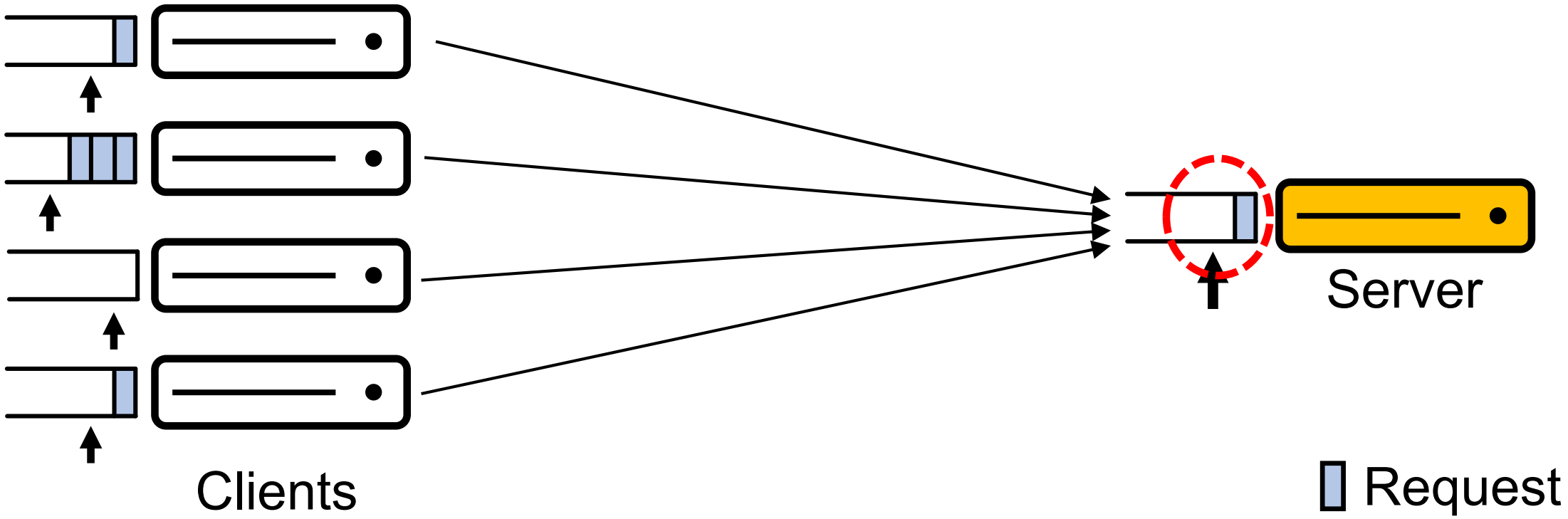
Overload Control Problem

We want to achieve **low latency** experienced by clients, and **high throughput** as long as latency isn't harmed.



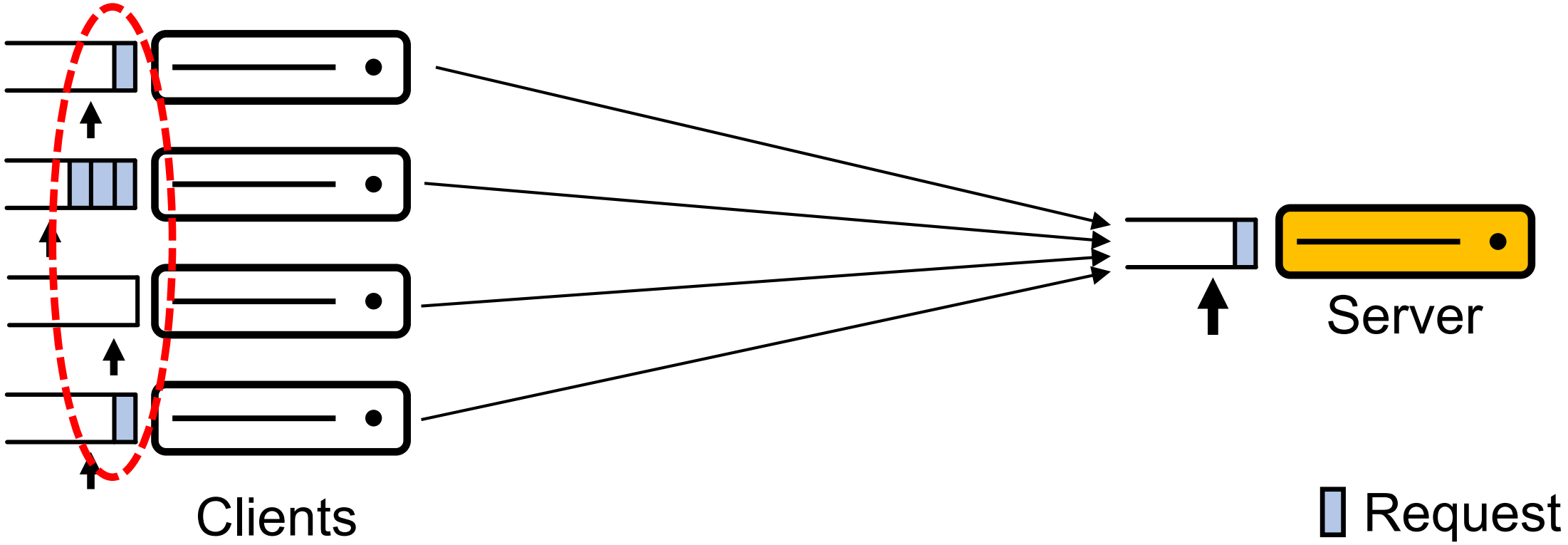
Ideal Overload Control

should keep request queue **short**, but **not empty**

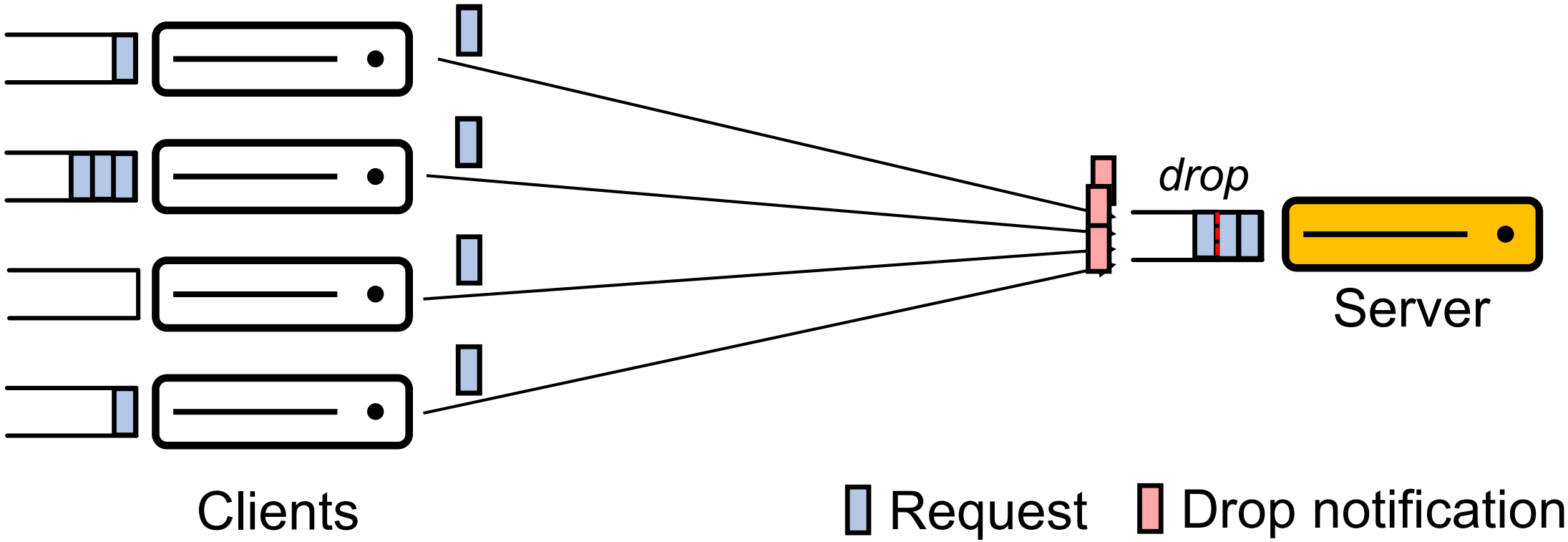


Ideal Overload Control

should inform clients about overload quickly

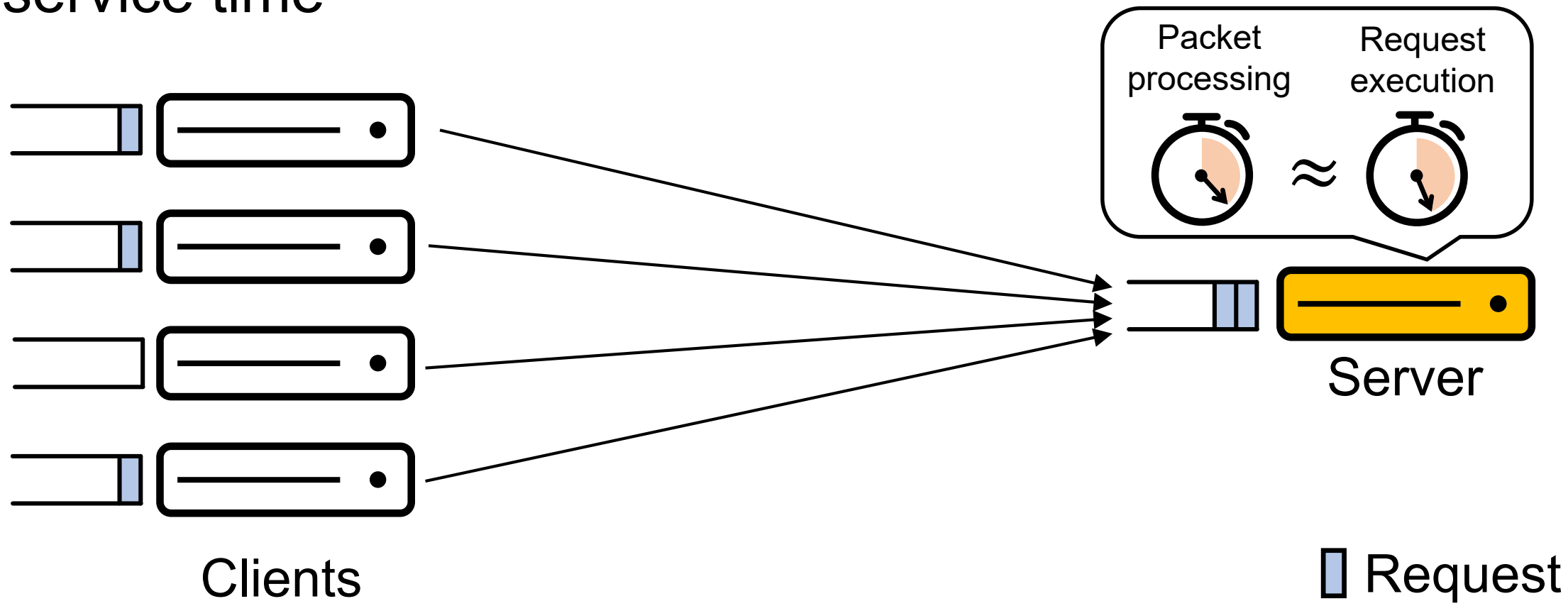


Strawman #1: Server-side AQM

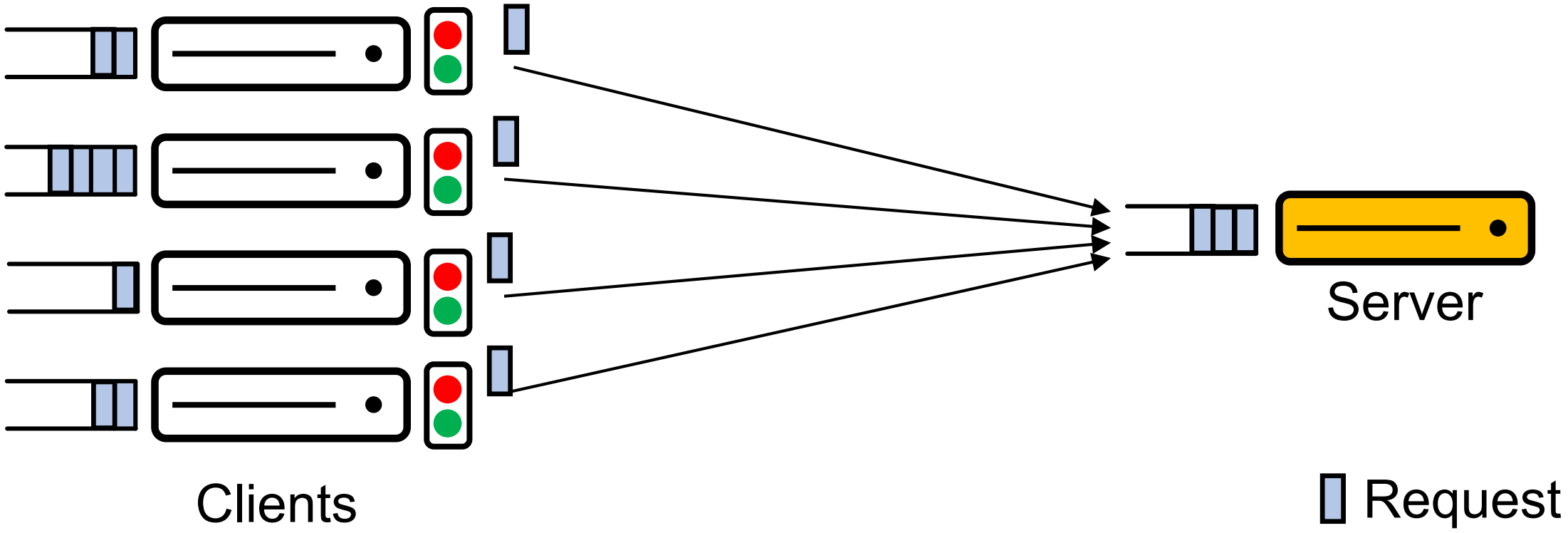


Strawman #1: Server-side AQM

The cost of packet processing is comparable to the service time

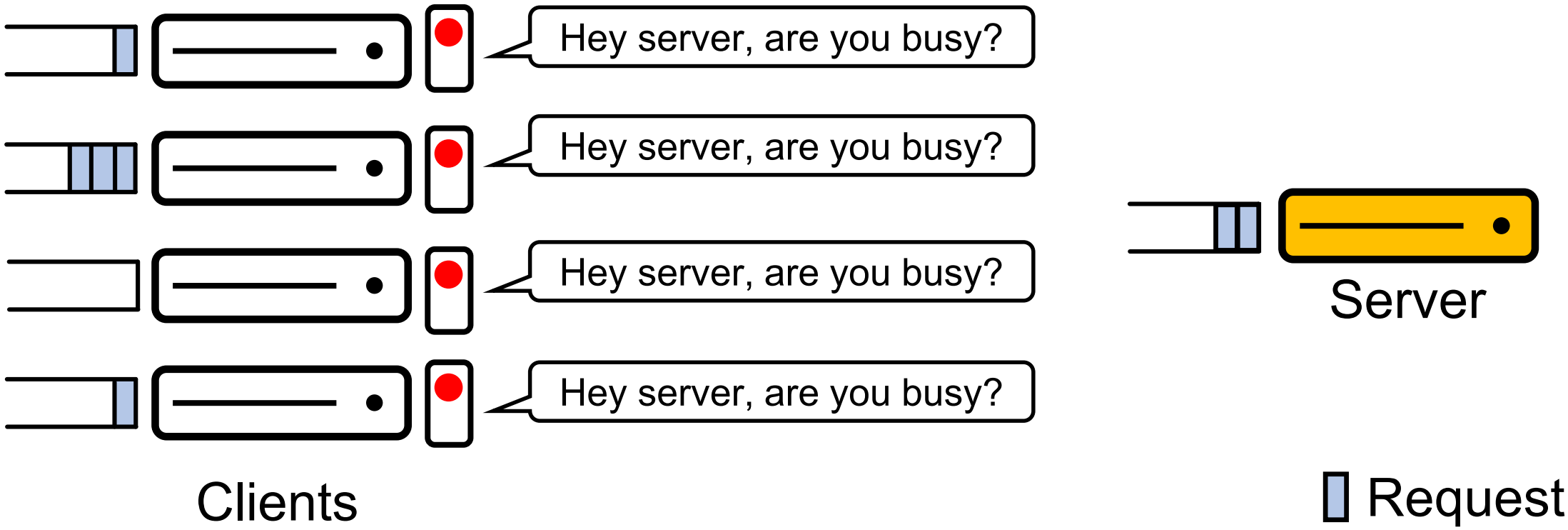


Strawman #2: Client Rate limiting

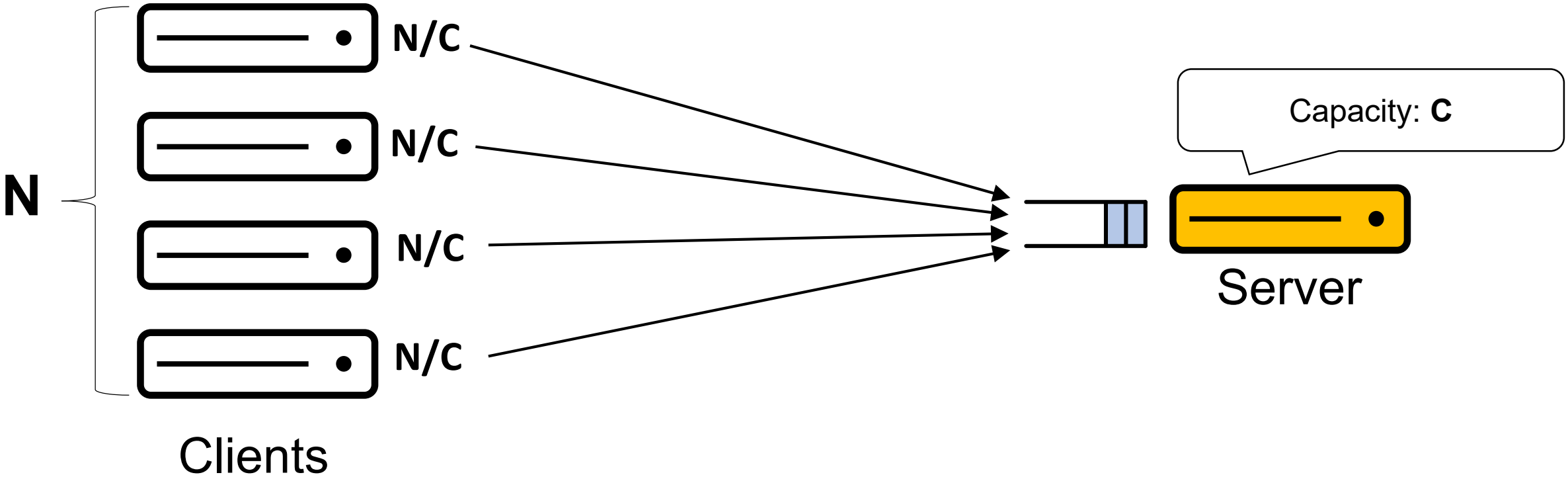


Strawman #2: Client Rate limiting

Probing server status incur high message overhead

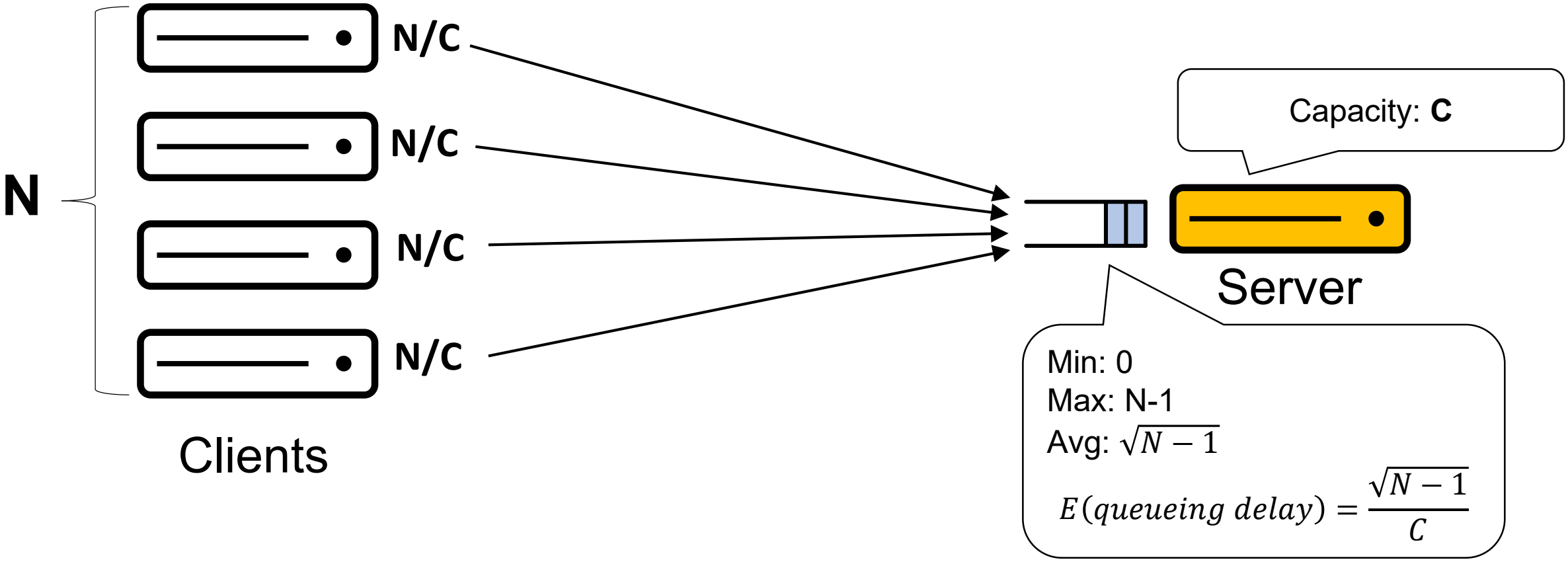


Strawman #2: Client Rate limiting



Strawman #2: Client Rate limiting

Even with clients' ideal rate, queue still can build up



Breakwater

Overload control scheme for µs-scale RPCs

Components	Benefits
1. Credit-based admission control	Coordinates requests with minimum delay
2. Demand speculation	Minimizes message overhead
3. Delay-based AQM	Ensures low tail latency

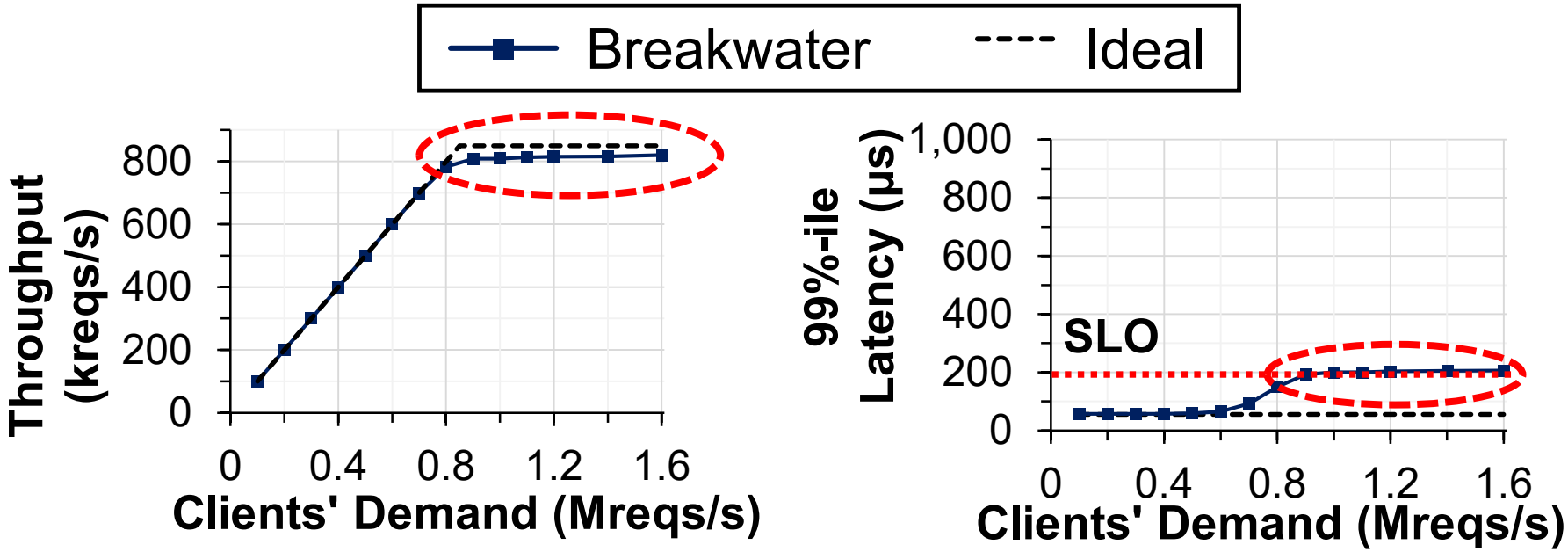
Breakwater's benefits

Handles server overload with **µs-scale RPCs** with

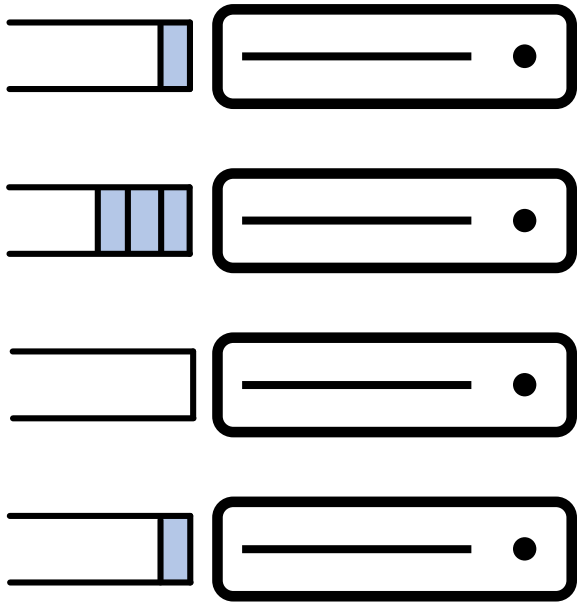
(1) High throughput

(2) Low and bounded tail latency

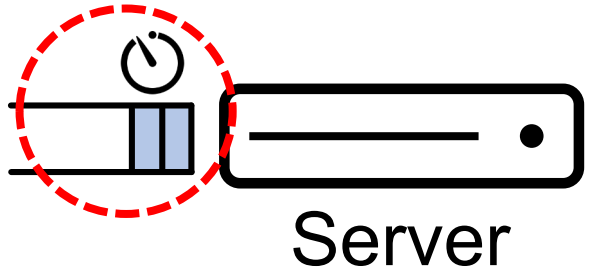
(3) Scalability to a large number of clients



Queueing delay as congestion signal



Clients

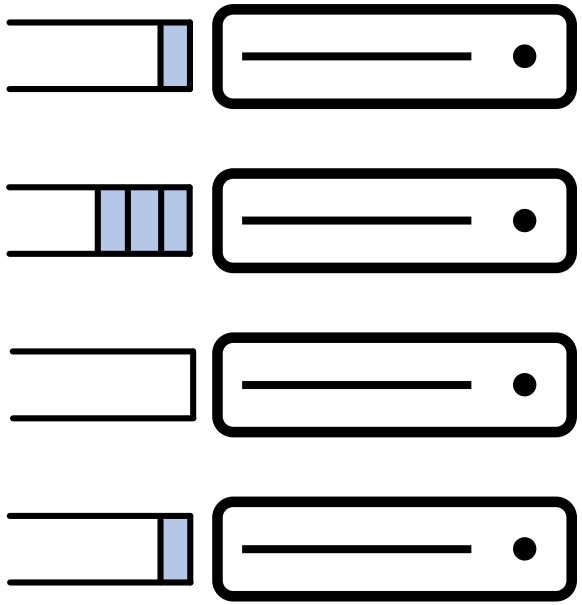


Server

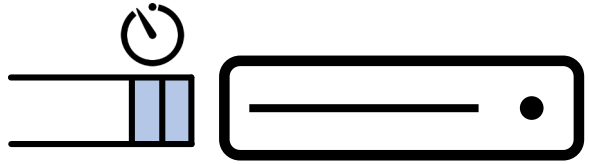
Credit Request Response

Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



Clients

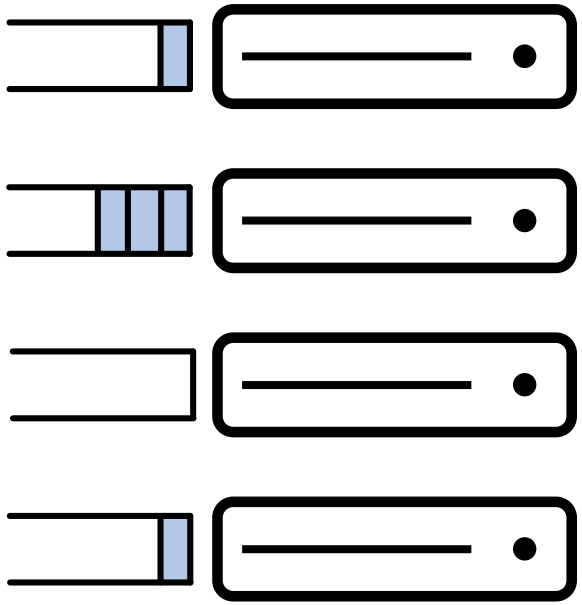


Server

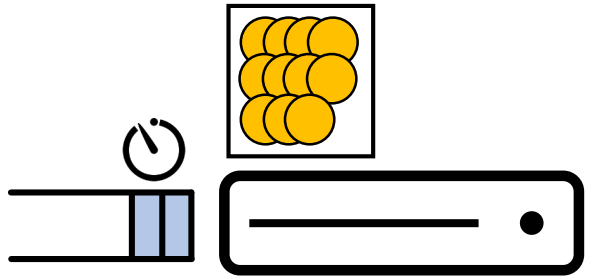
Credit Request Response

Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



Clients

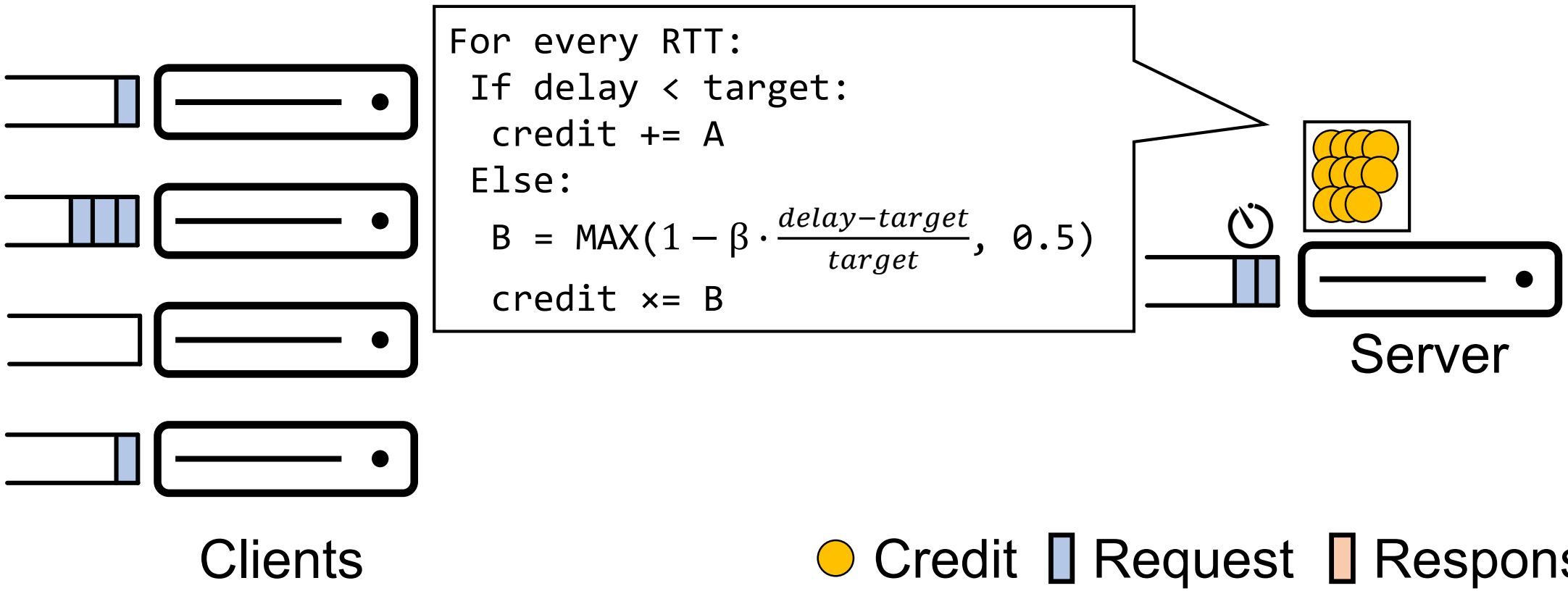


Server

Credit Request Response

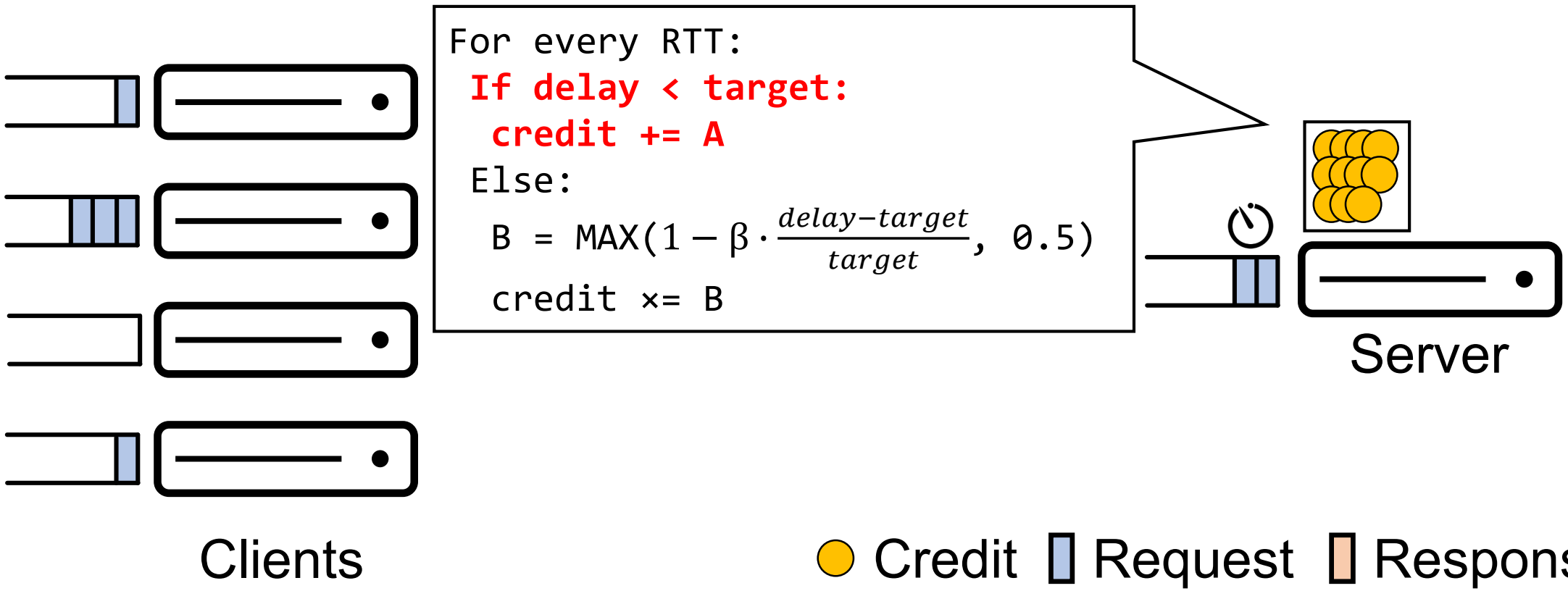
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



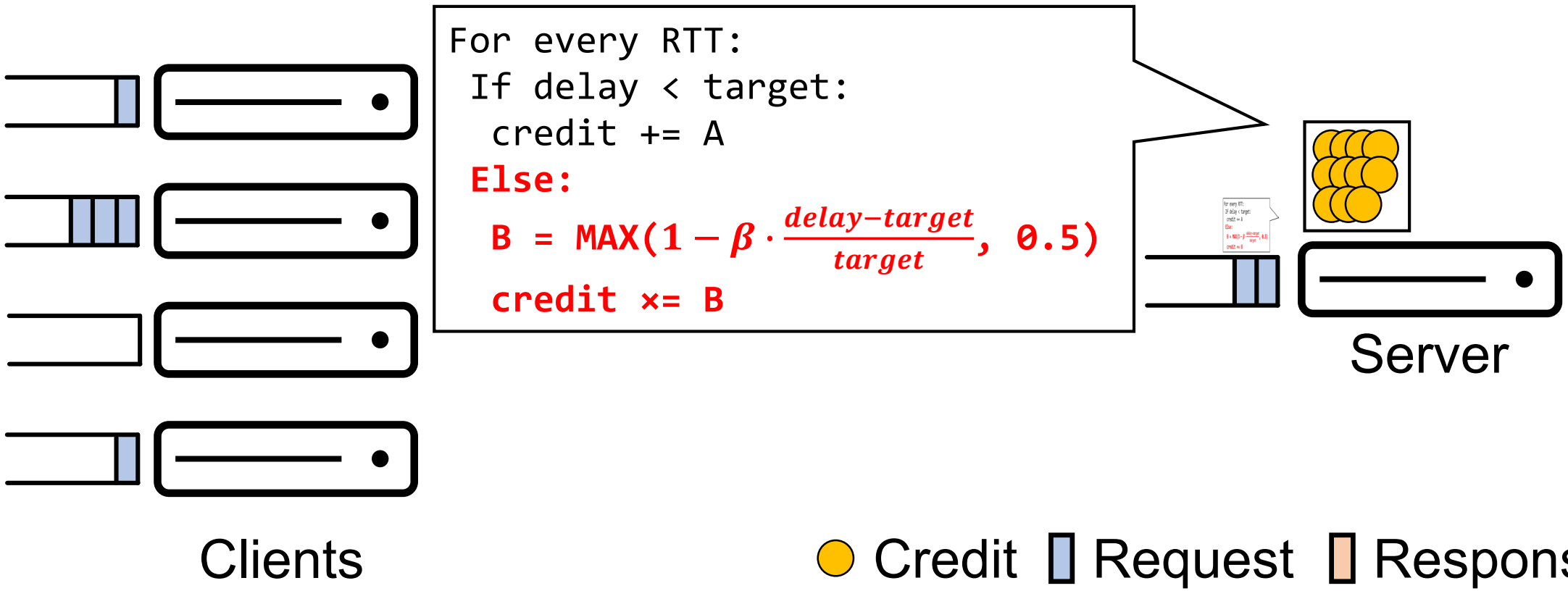
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



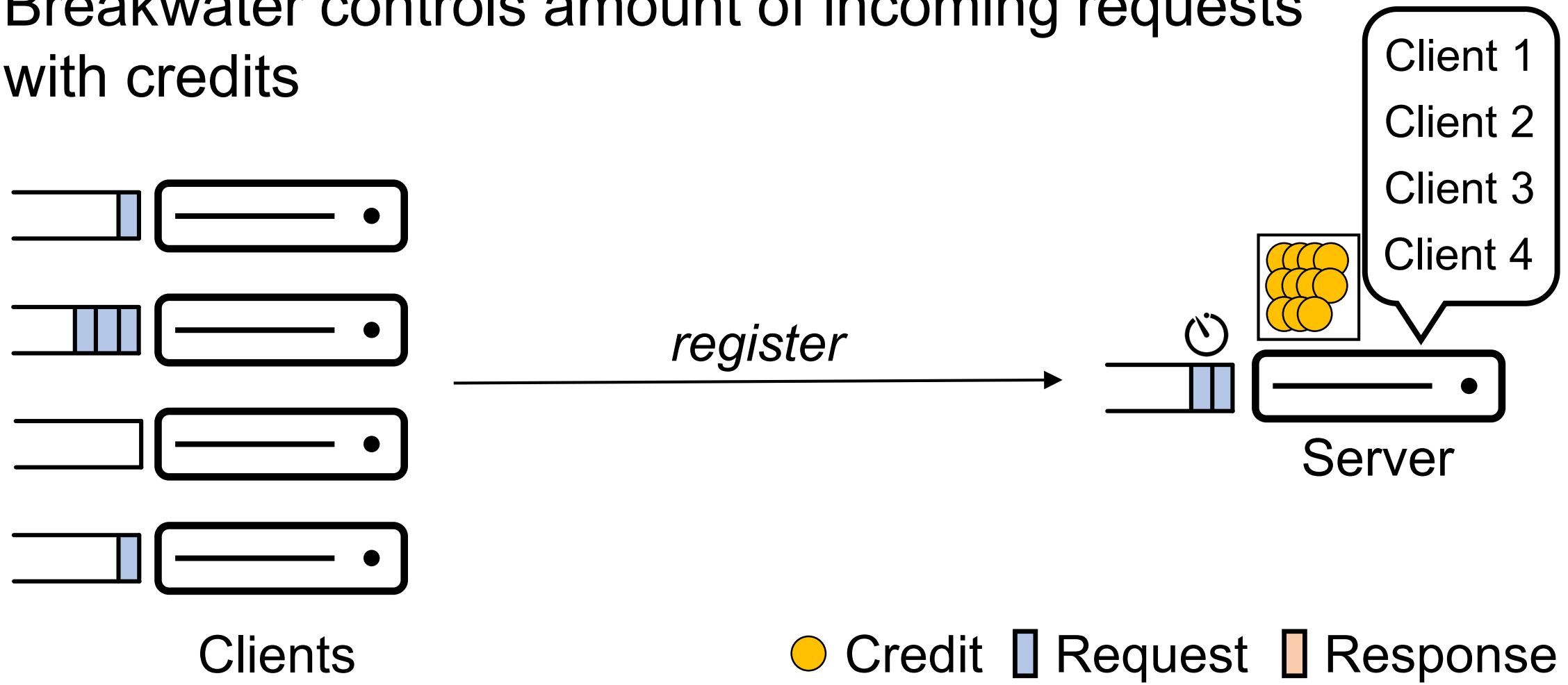
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



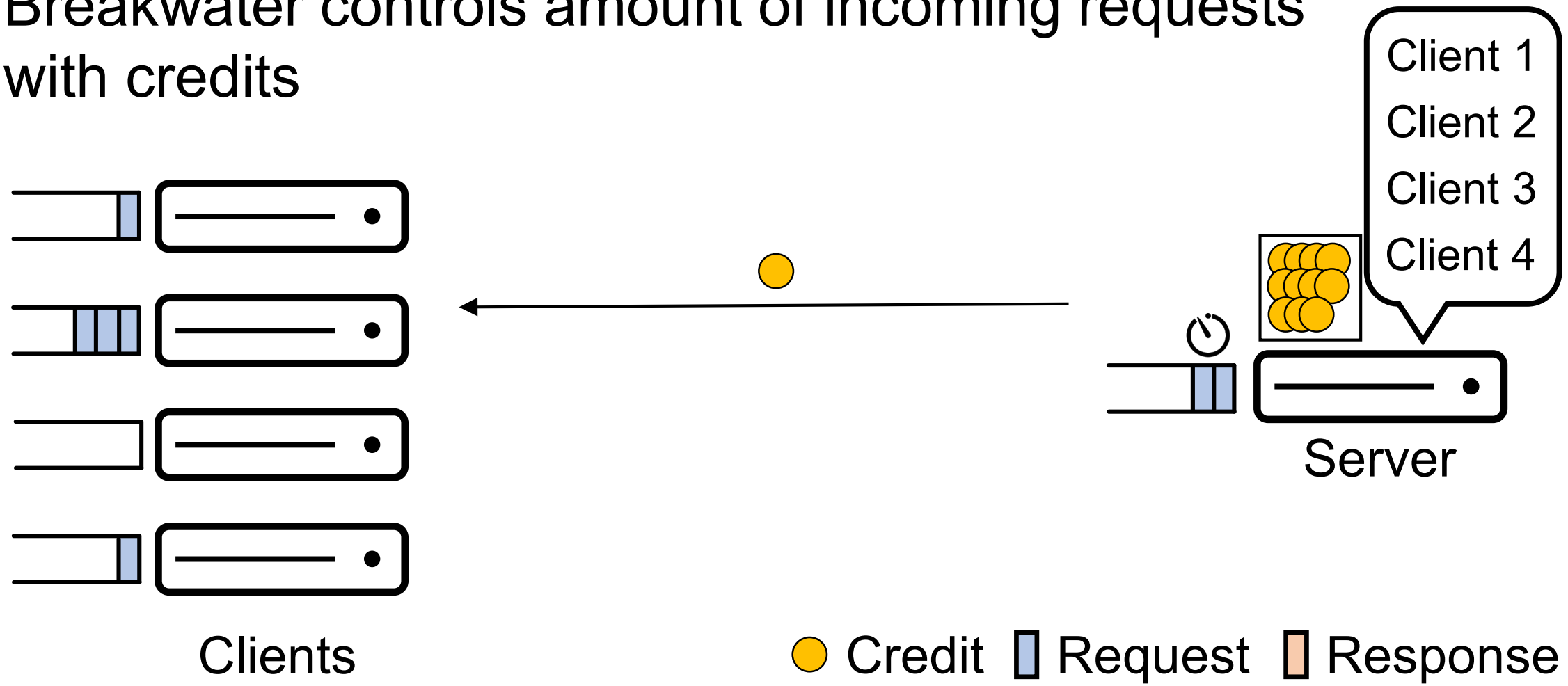
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



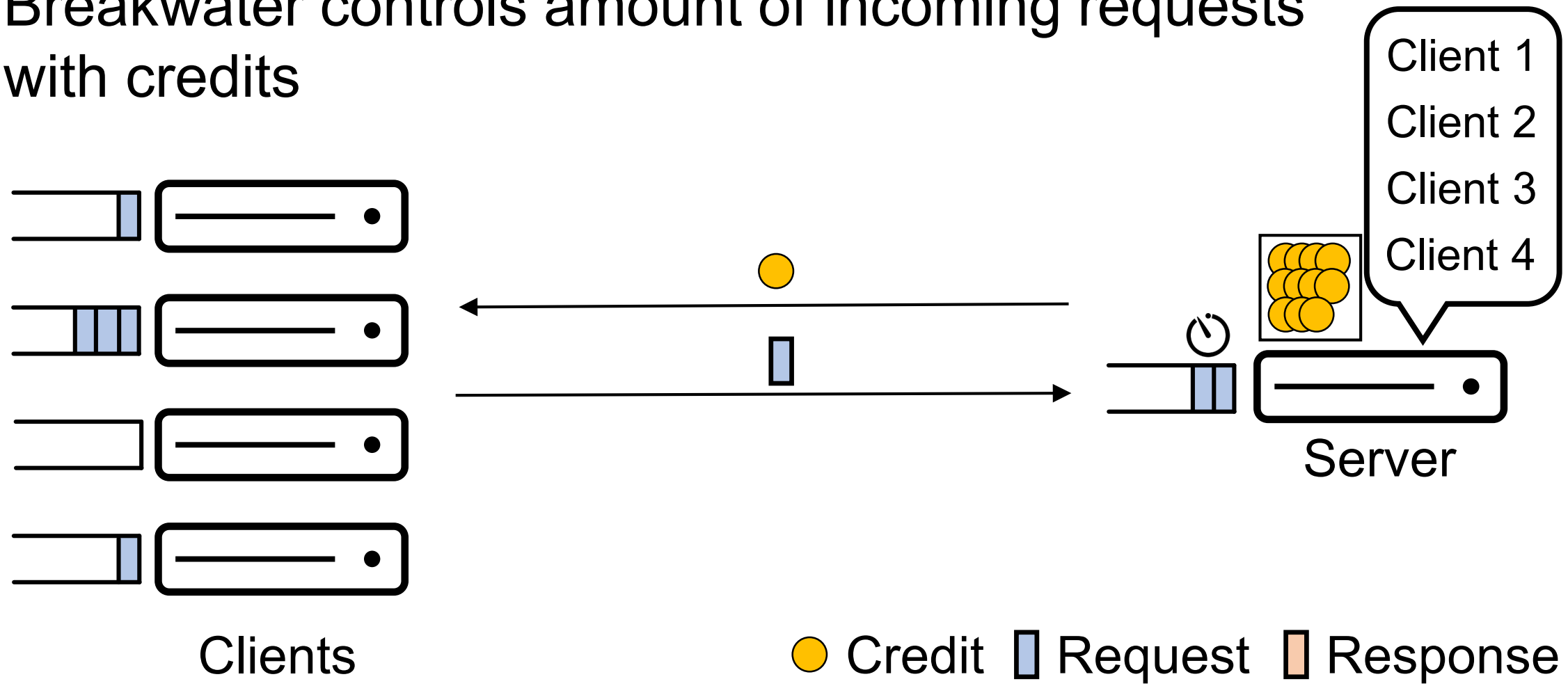
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



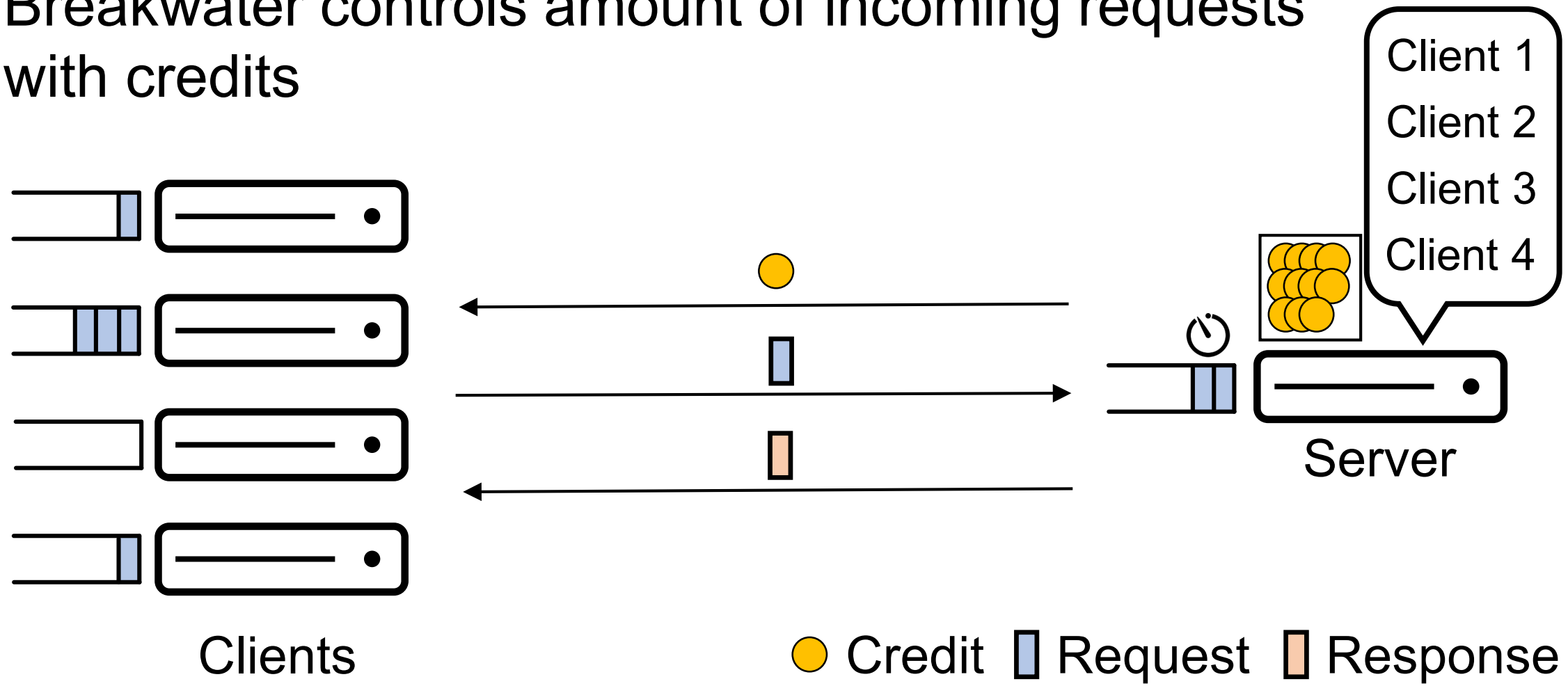
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



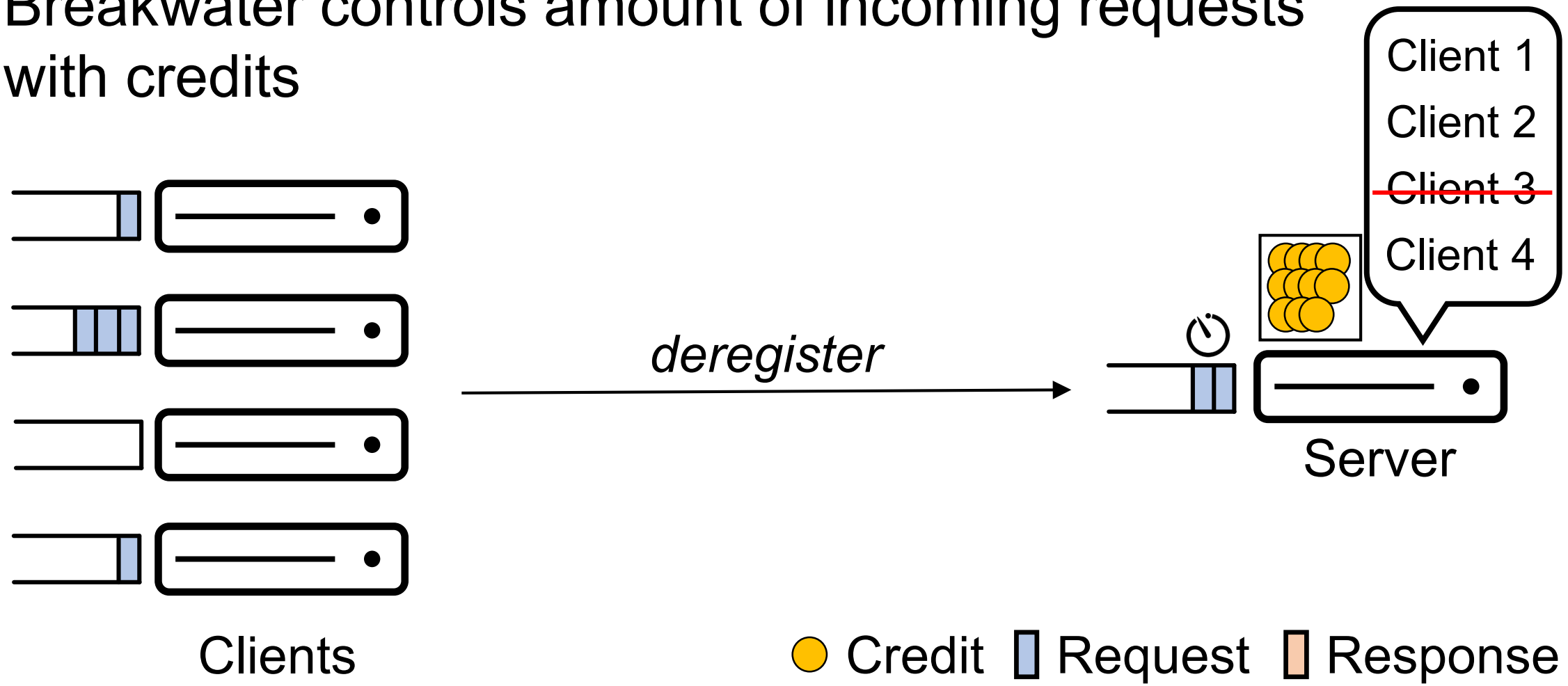
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



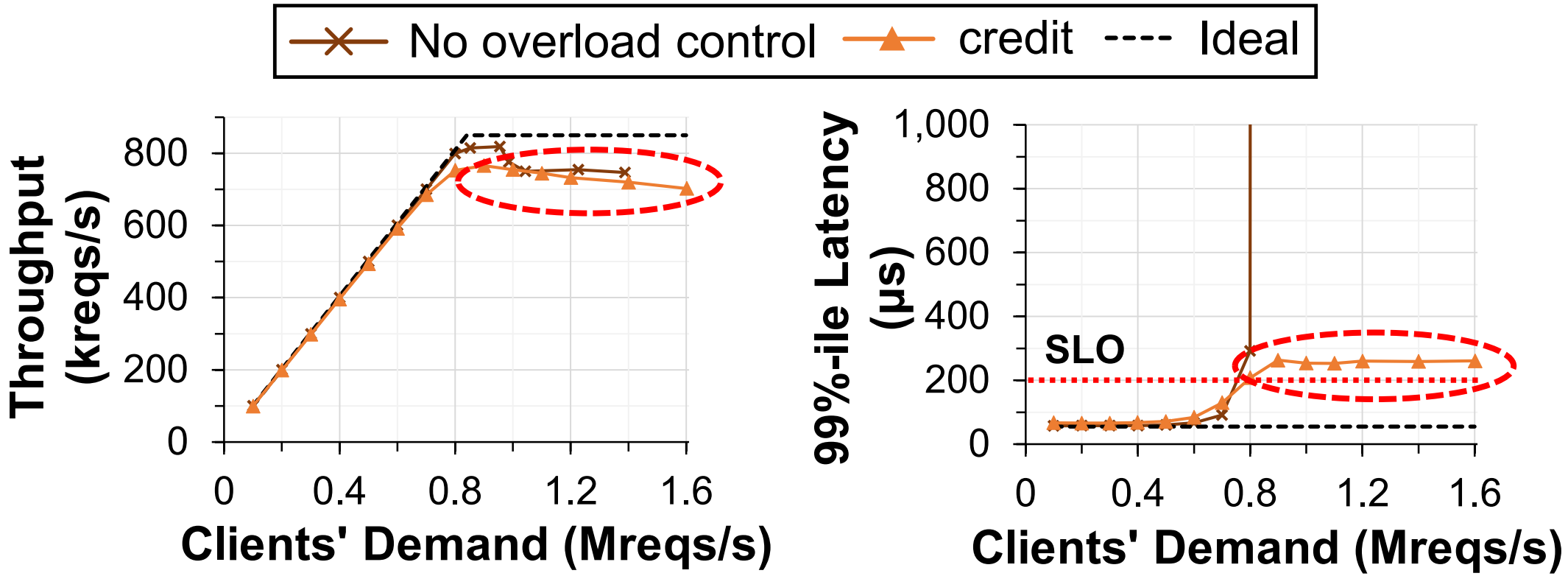
Comp. #1: Credit-based admission control

Breakwater controls amount of incoming requests with credits



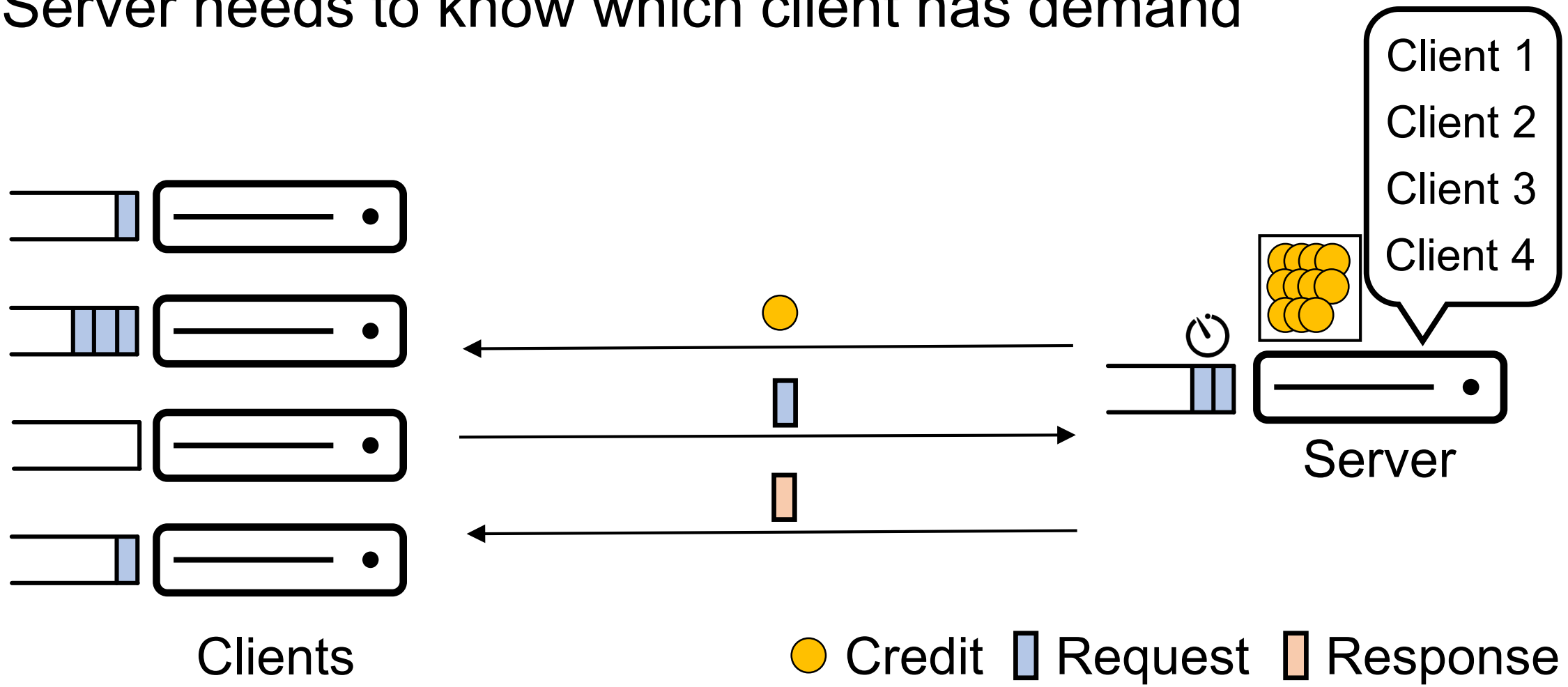
Impact of Credit-based Admission Control

Credit-based admission control has lower and bounded tail latency but lower throughput.



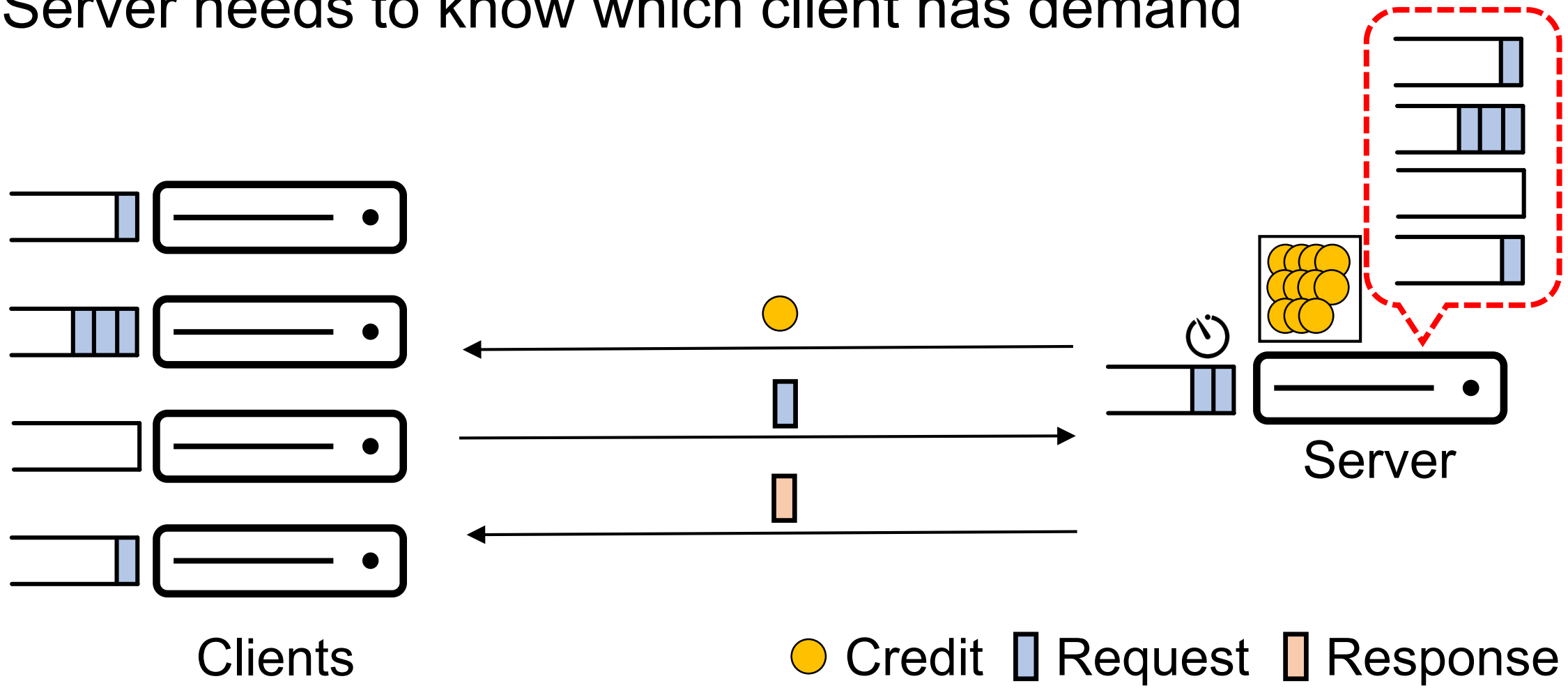
Demand Message Overhead

Server needs to know which client has demand



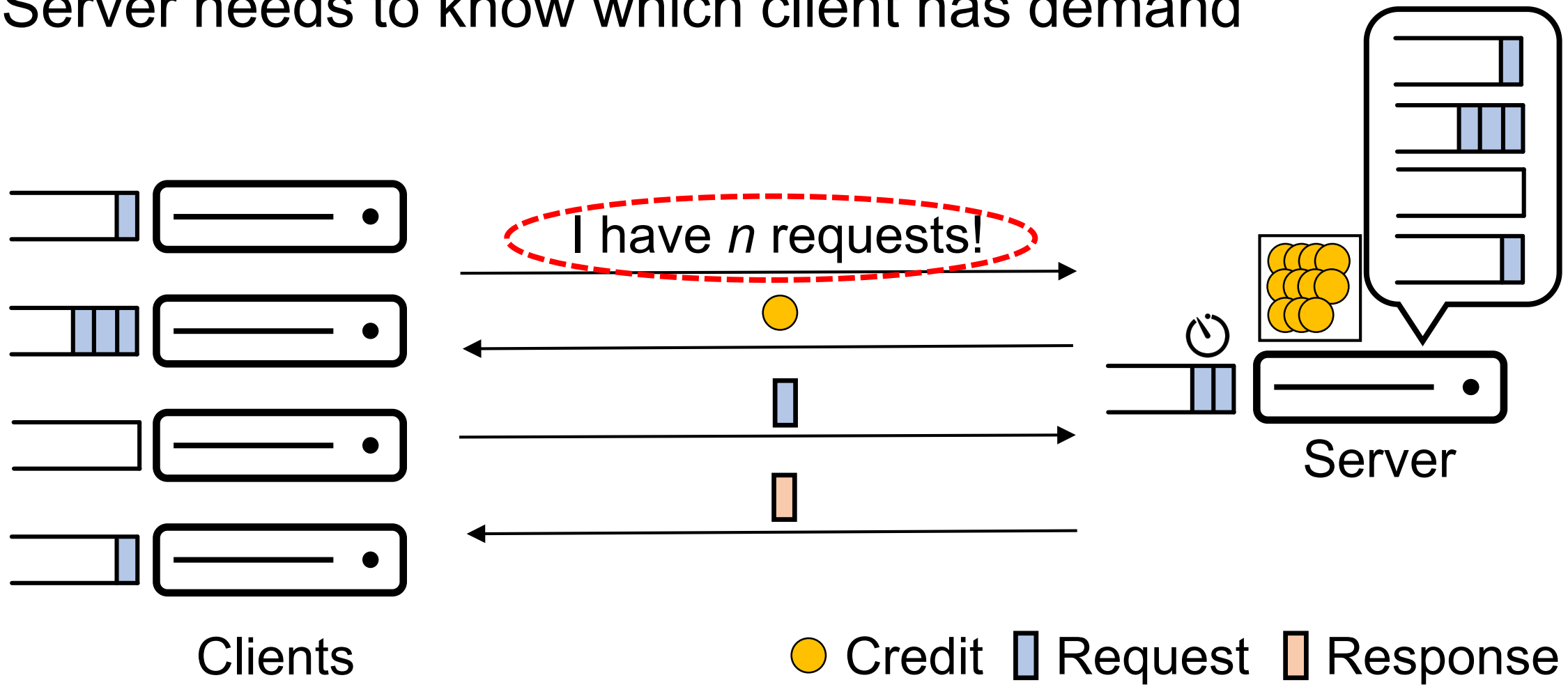
Demand Message Overhead

Server needs to know which client has demand



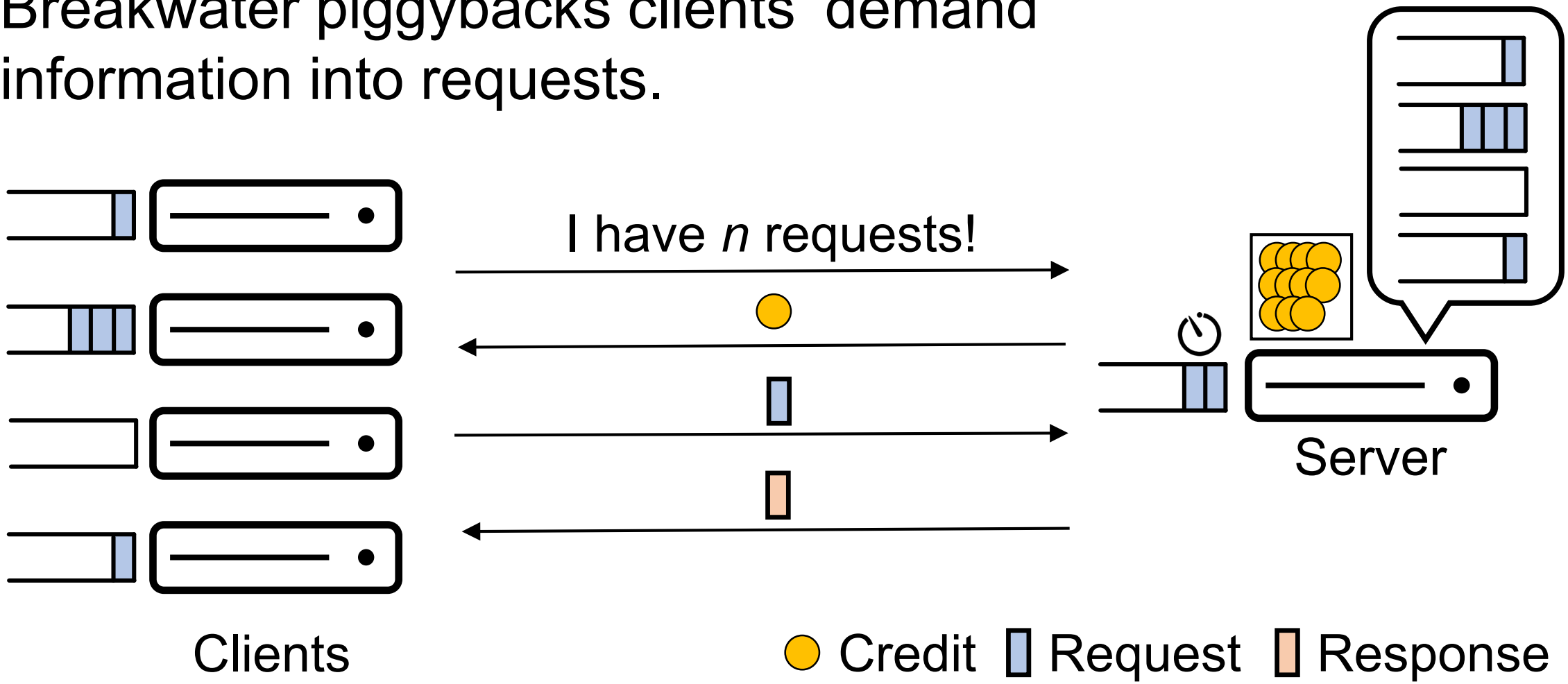
Demand Message Overhead

Server needs to know which client has demand



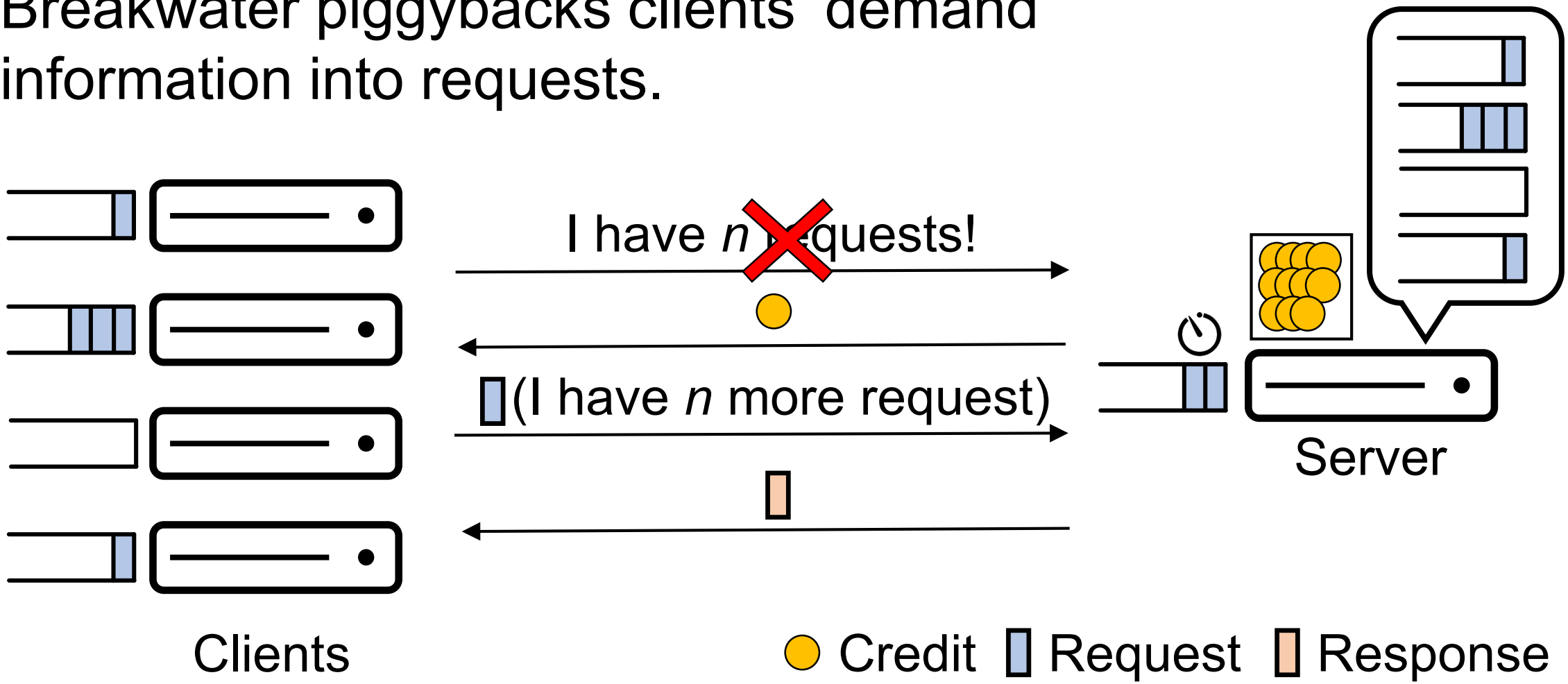
Piggybacking Demand Information

Breakwater piggybacks clients' demand information into requests.



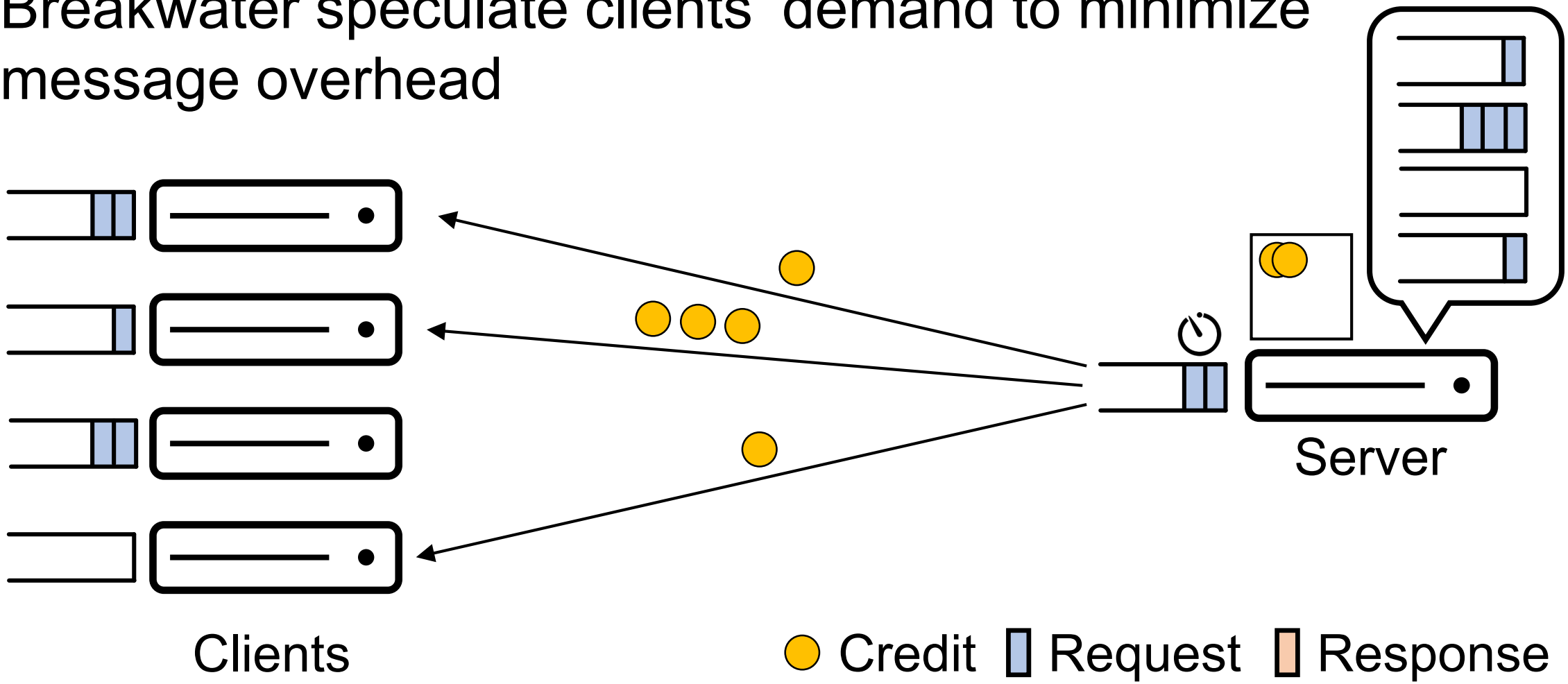
Piggybacking Demand Information

Breakwater piggybacks clients' demand information into requests.



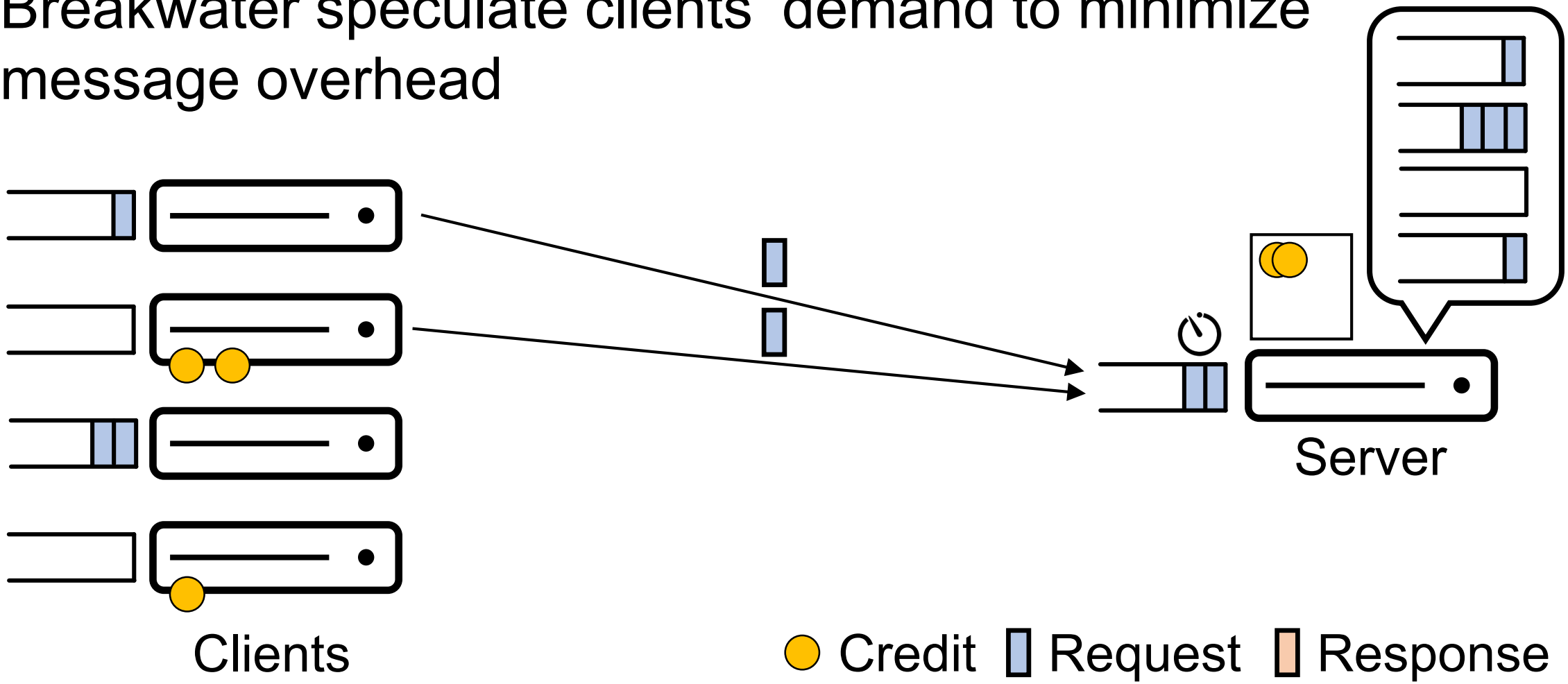
Comp. #2: Demand Speculation

Breakwater speculate clients' demand to minimize message overhead



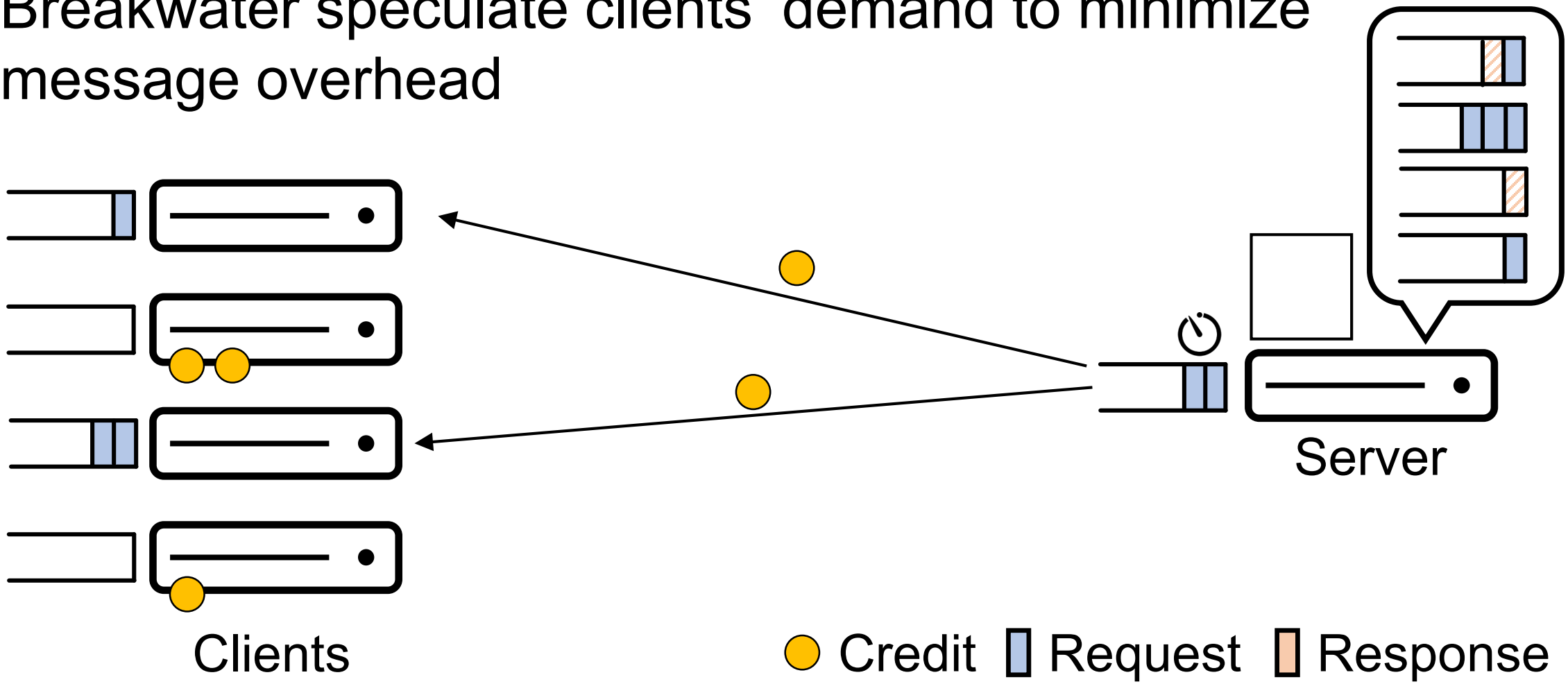
Comp. #2: Demand Speculation

Breakwater speculate clients' demand to minimize message overhead



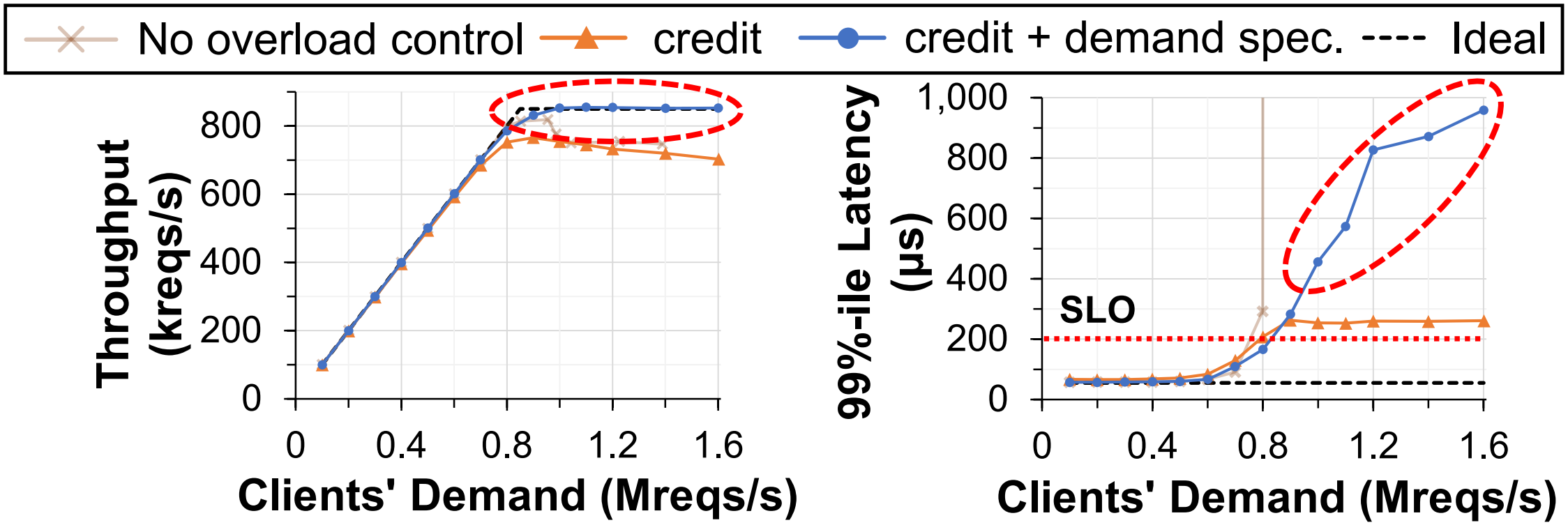
Comp. #2: Demand Speculation

Breakwater speculate clients' demand to minimize message overhead



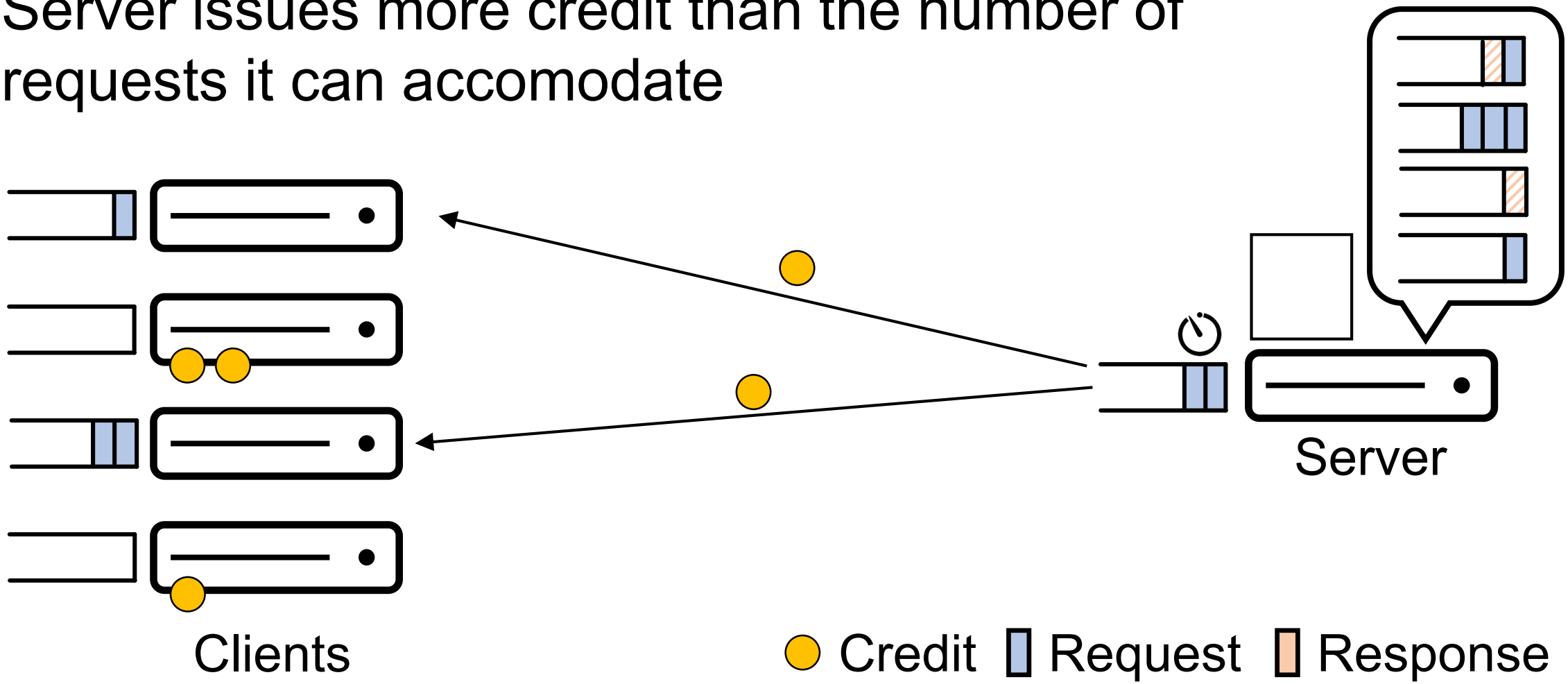
Impact of Adding Demand Speculation

Demand speculation improves throughput with higher tail latency



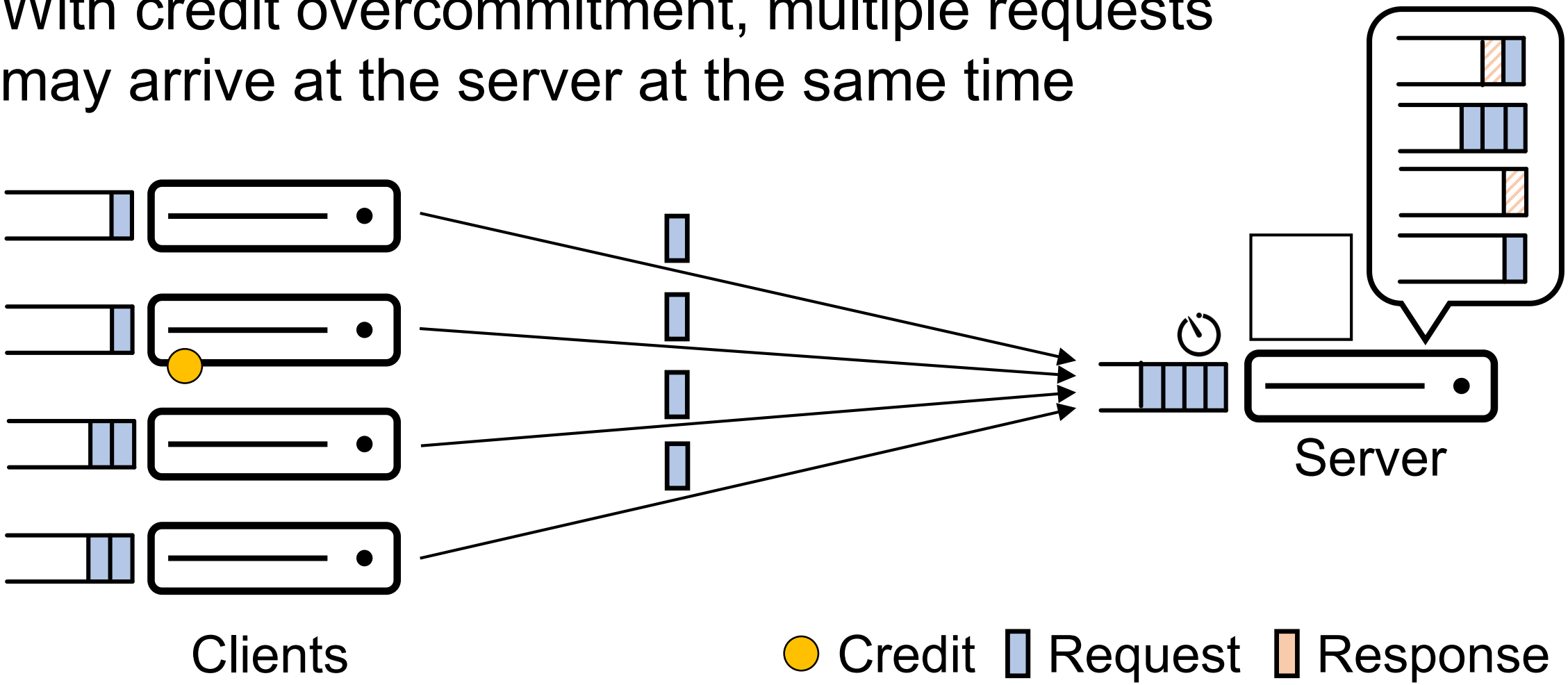
Credit Overcommitment

Server issues more credit than the number of requests it can accomodate



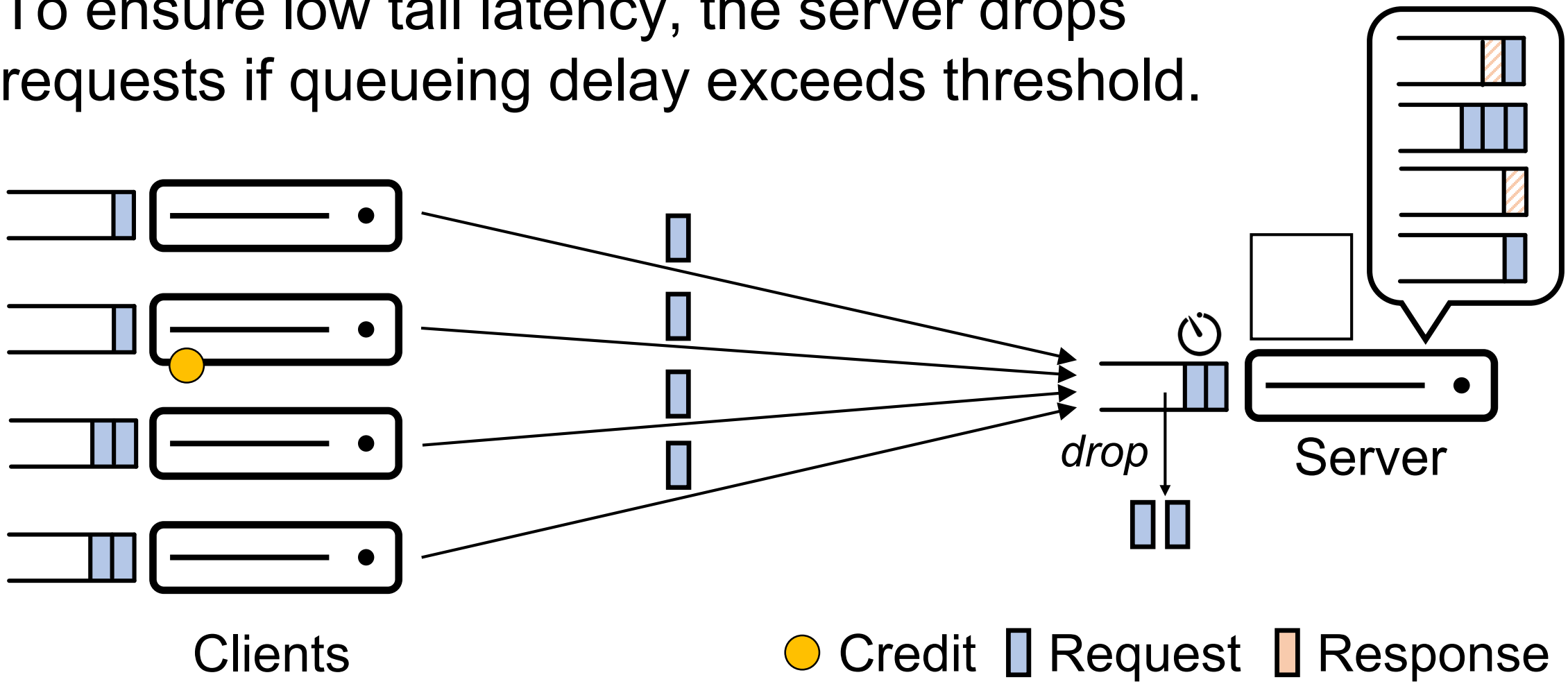
Incast Causing Long Queue

With credit overcommitment, multiple requests may arrive at the server at the same time



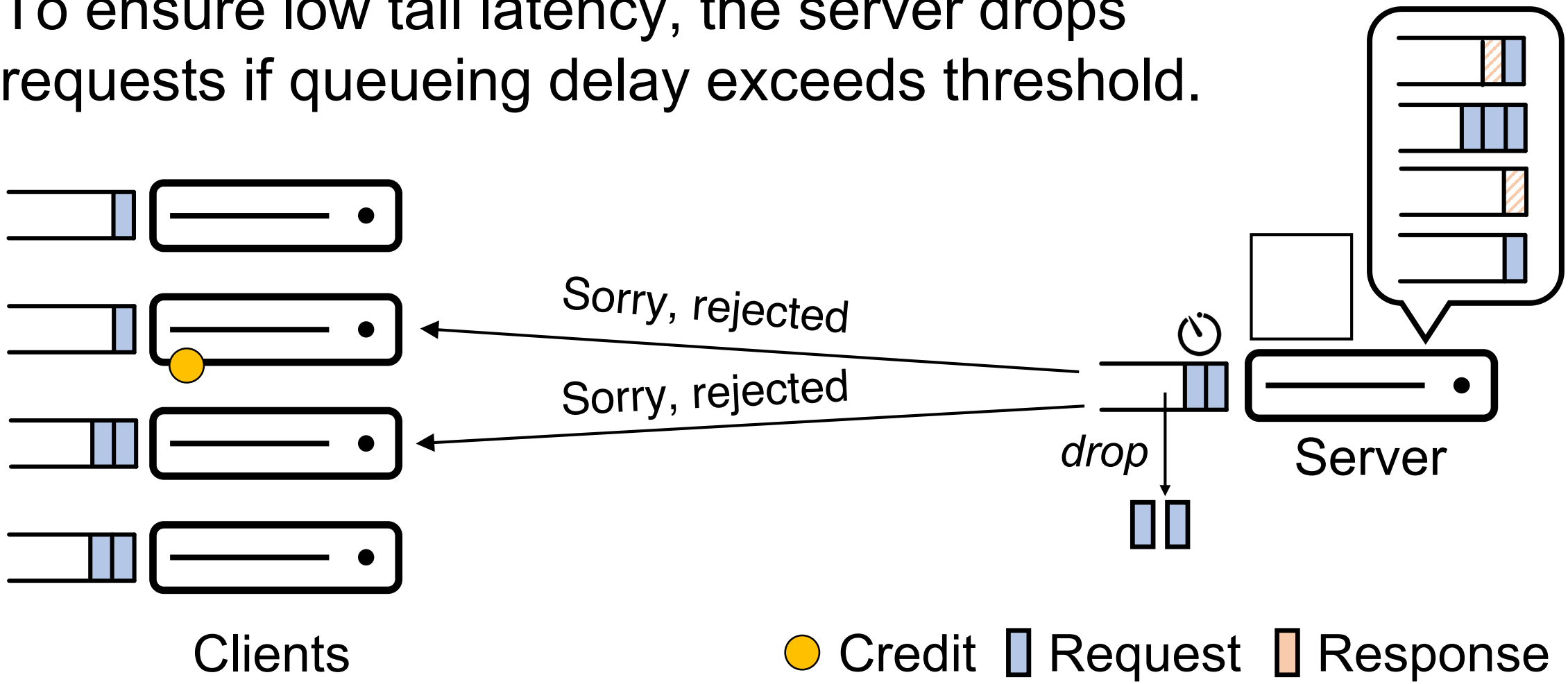
Comp. #3: Delay-based AQM

To ensure low tail latency, the server drops requests if queueing delay exceeds threshold.



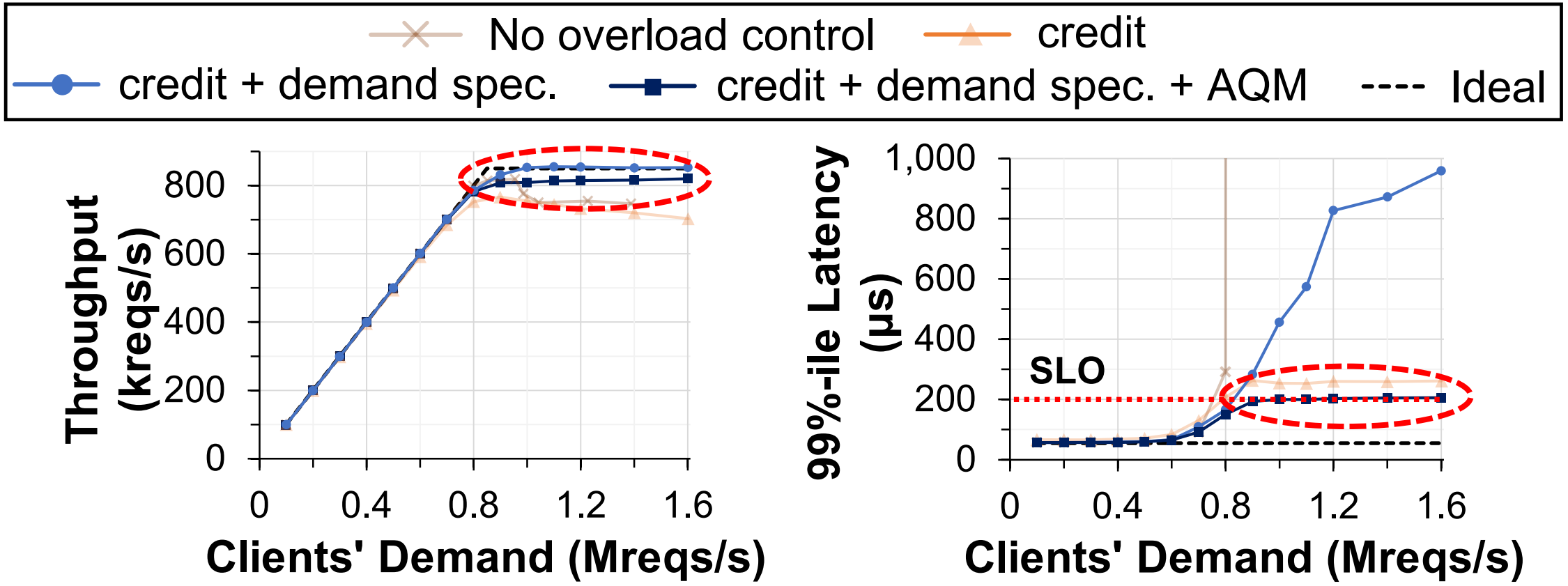
Comp. #3: Delay-based AQM

To ensure low tail latency, the server drops requests if queueing delay exceeds threshold.



Impact of Adding Delay-based AQM

Breakwater achieves high throughput and low and bounded tail latency at the same time



Evaluation

Testbed Setup

- xl170 in Cloudlab
- 11 machines are connected to a single switch
- 10 client machines / 1 server machine
- Implementation on Shenango as a RPC layer

Synthetic Workload

- Clients generate request with open-loop Poisson process
- Requests spin-loops specified amount of time at server
- Exponential service time distribution with 10µs average

Evaluation

(1) Does Breakwater achieves high throughput and low tail latency even with demand spikes?
(2) Does Breakwater provides fast notification for the rejected requests?
(3) Is Breakwater scalable to many clients?

Baselines:

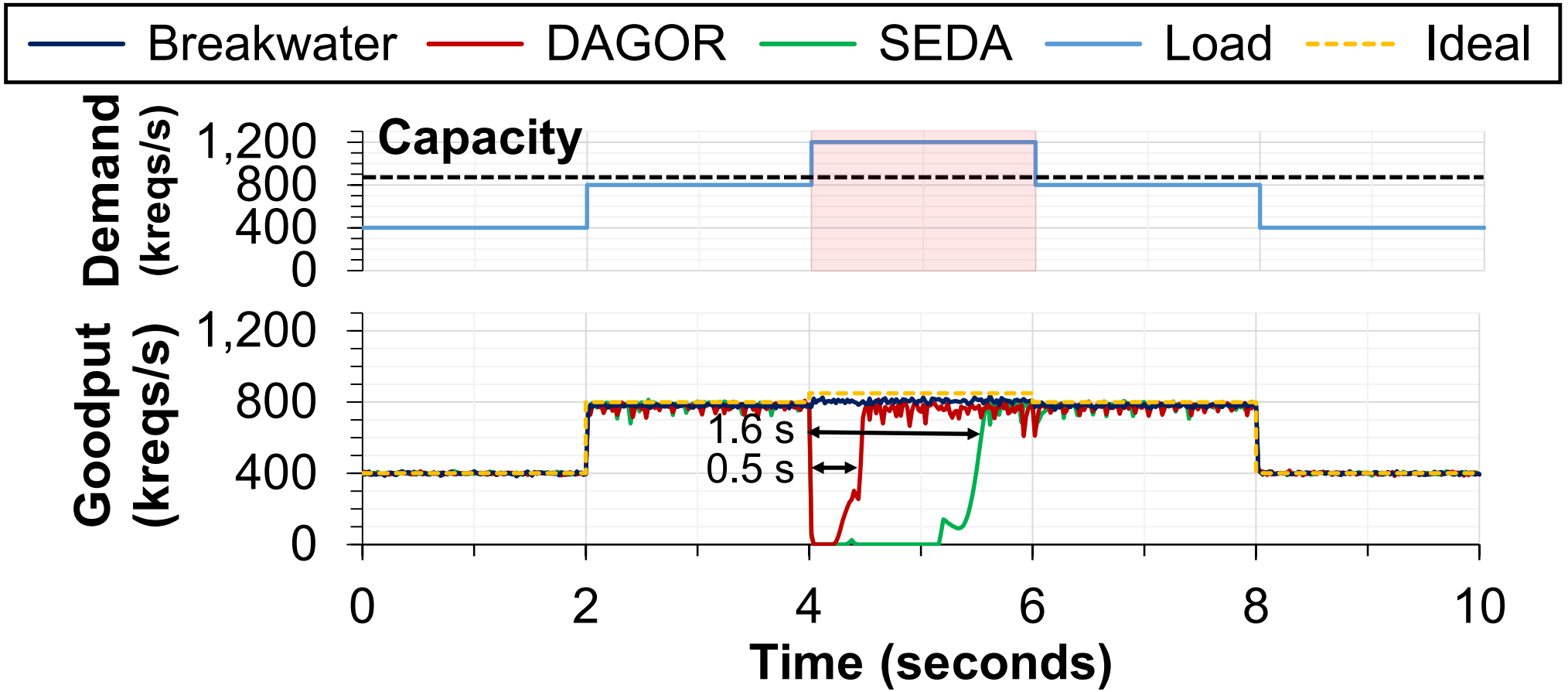
DAGOR

priority-based overload control used in WeChat

SEDA

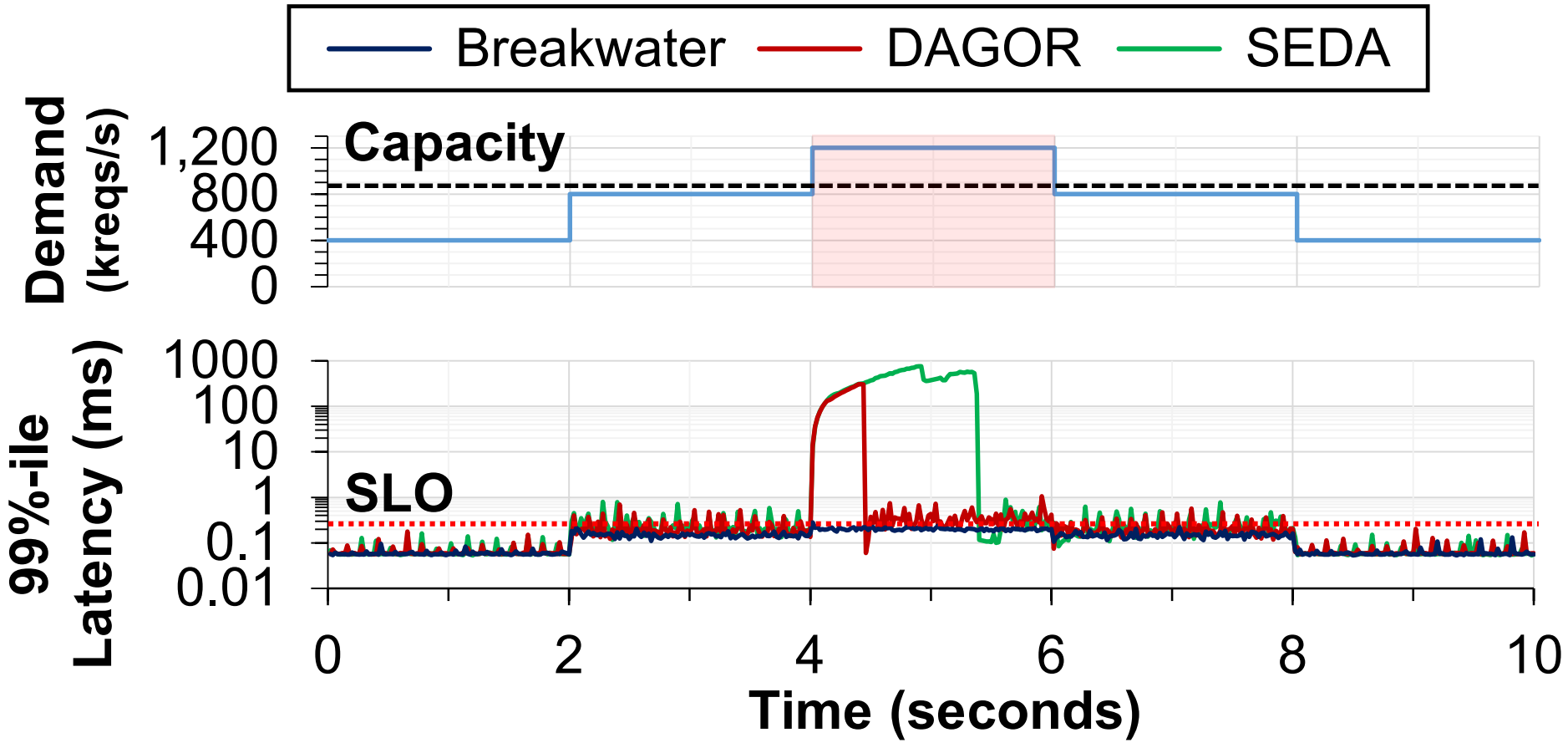
adaptive overload control for staged event-driven architecture

High Goodput with Fast Convergence



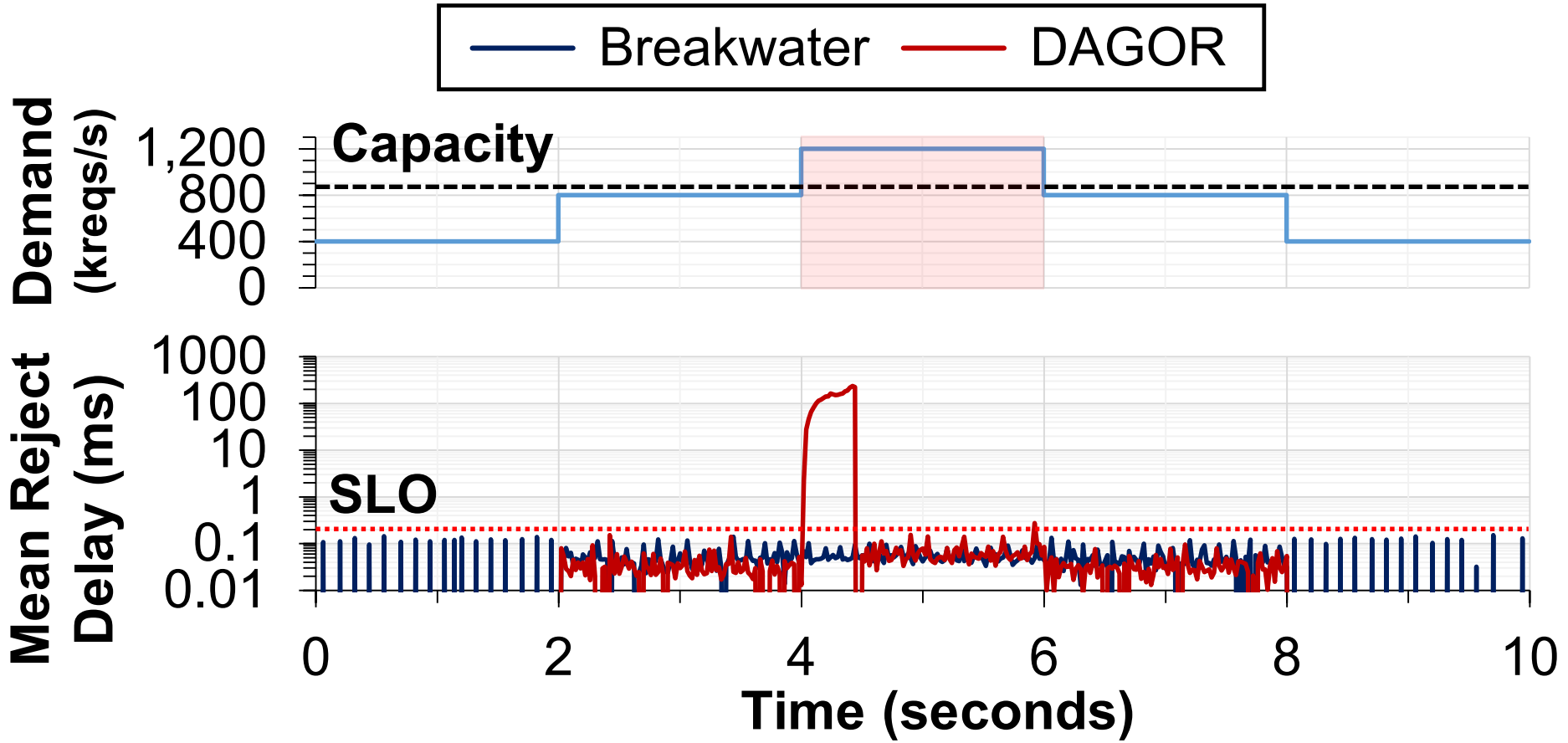
Breakwater converges to higher goodput 25x faster than DAGOR and 79x faster than SEDA.

Low and Bounded Tail Latency



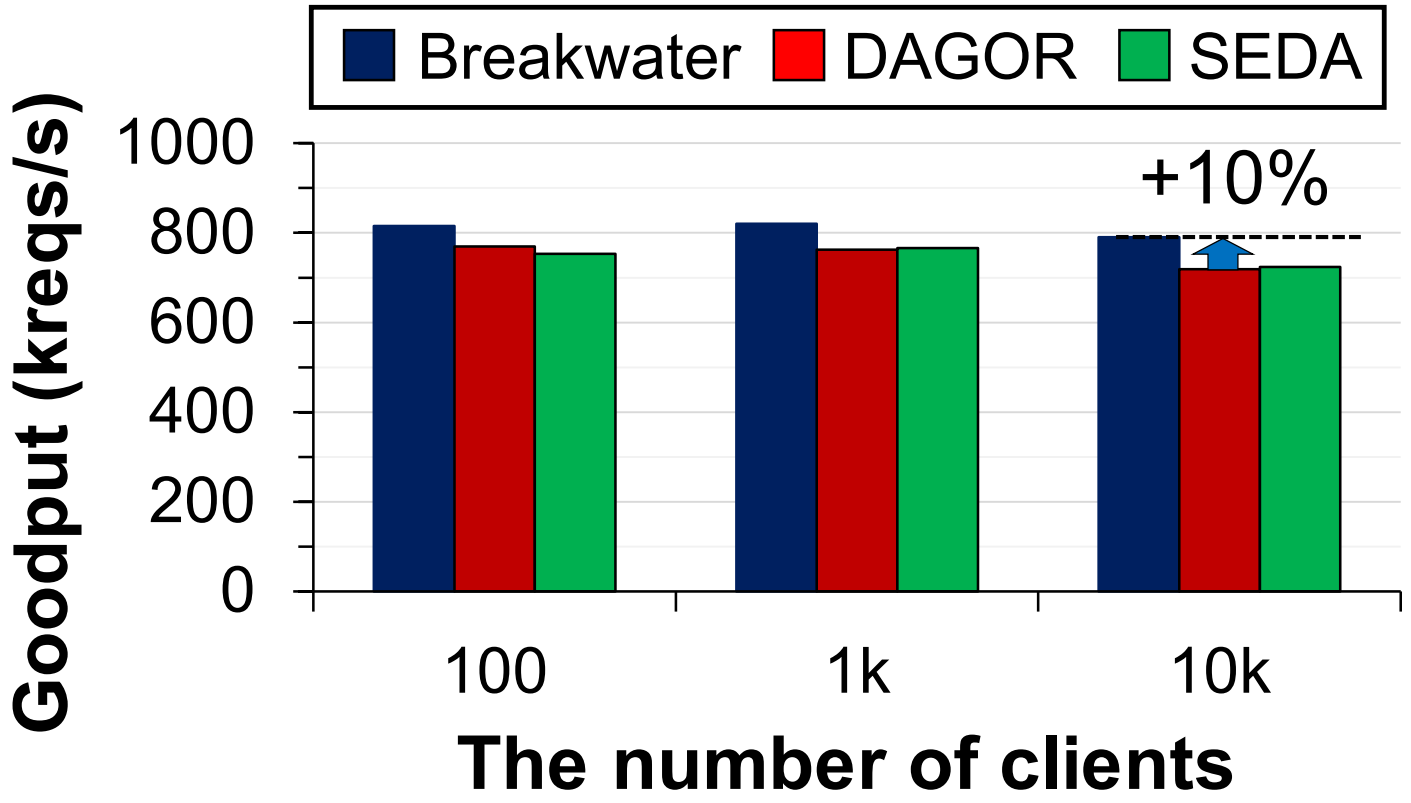
Breakwater maintains low tail latency even with load spikes.

Fast Notification of Reject



Breakwater notifies rejected request to clients before violating its SLO.

Scalability to Many Clients



Breakwater easily scales to 10,000 clients.

Conclusion

- Breakwater is a **server-driven credit-based** overload control system for **µs-scale RPCs**
- Breakwater's key components include
  (1) Credit-based admission control
  (2) Demand speculation
  (3) Delay-based AQM
- Our evaluation shows that Breakwater achieves
  (1) **Low & bounded tail latency** with **high throughput**
  (2) **Fast notification** for a rejected request
  (3) **Scalability** to many clients

Open Questions

- Can you think of better congestion signal?
- What if other than CPU is the bottleneck?
- What if we use processor sharing model instead of run-to-completion?
- What would be the corner case where Breakwater is not efficient? Can you make it more efficient?
- What infrastructure can you build on top of Breakwater?

Feel free to play around with it!

Github:
http://inhocho89.github.io/breakwater

Full paper:
https://inhocho89.github.io/papers/osdi20overload.pdf

Intel PT

Inho Cho

Intel-PT

Suppose you want to record all the instruction executed by the CPU

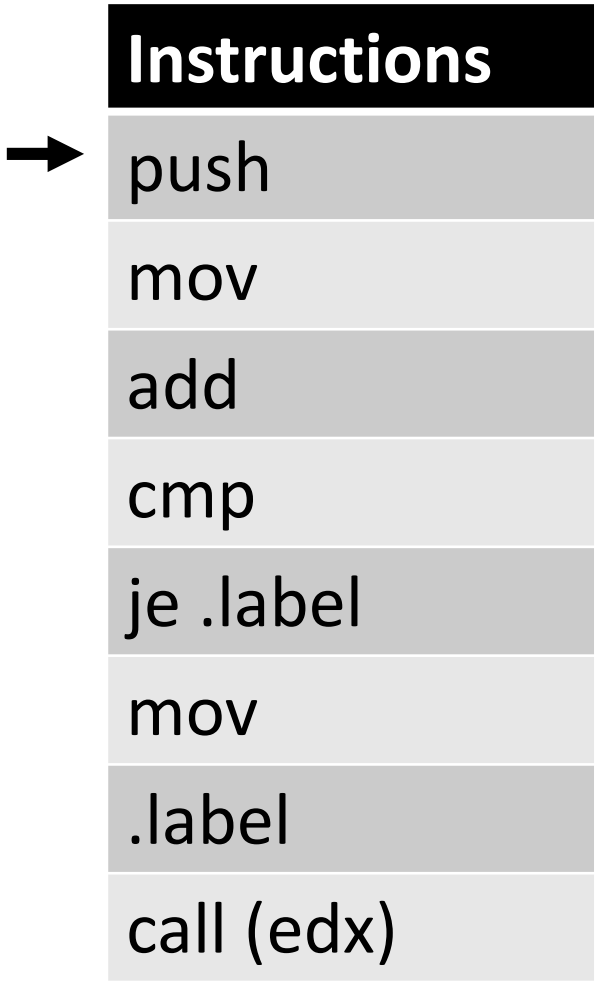
Hardware supported Process Trace

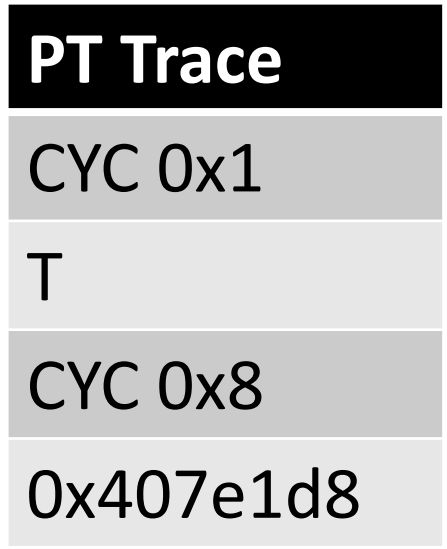
- It records whether a branch is Taken(T)/Not Taken (NT) for “every” branch during program execution

Intel-PT Example

Instructions
push
mov
add
cmp
je .label
mov
.label
call (edx)

Intel-PT Example

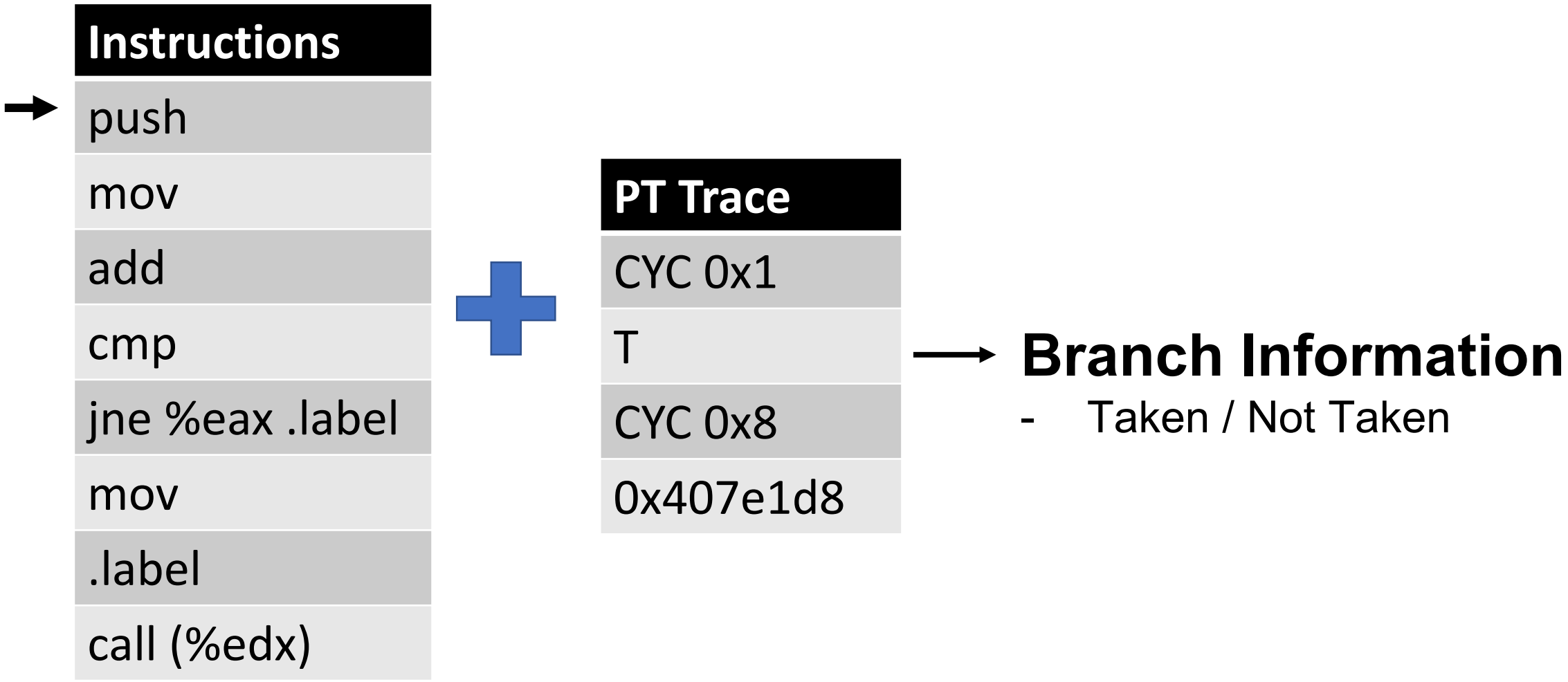




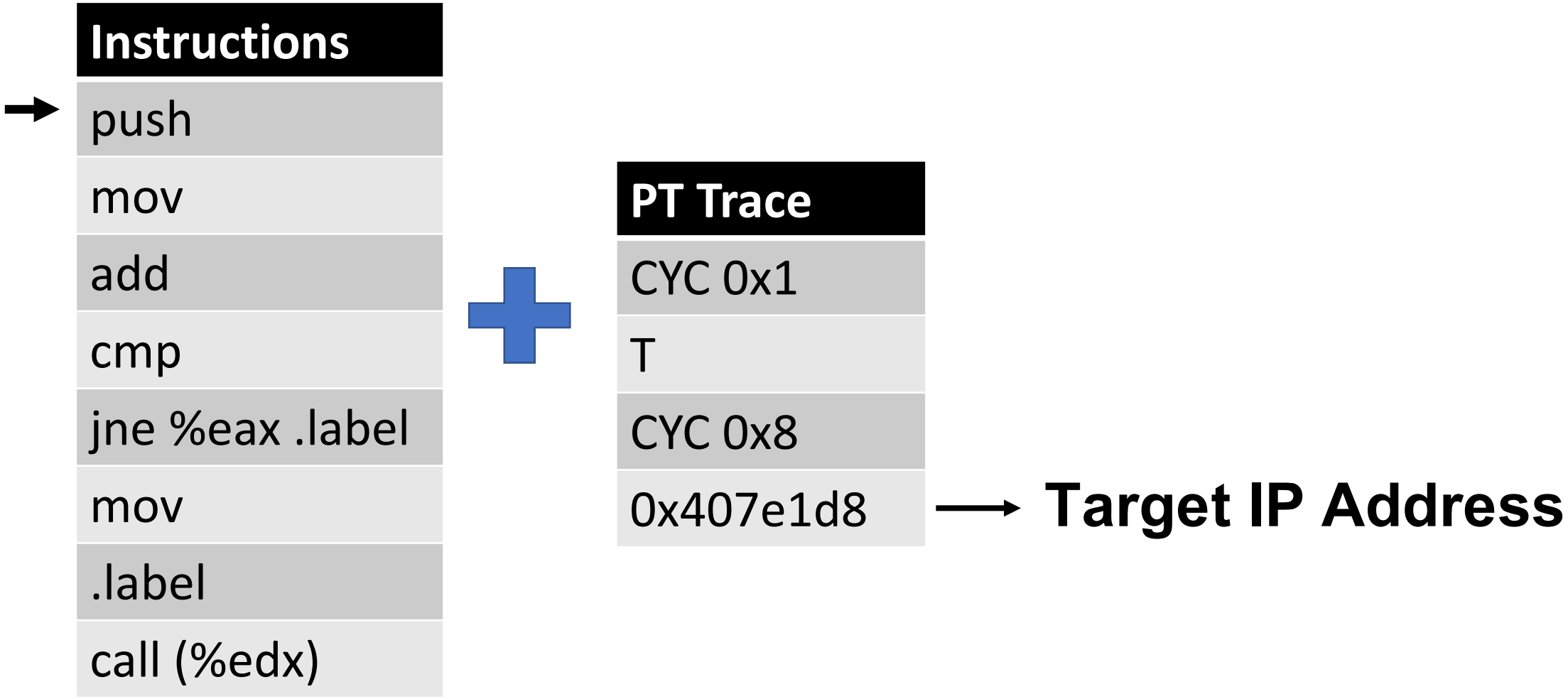
Timing Information

- PSB: TSC
- [Default] MTC: from hardware crystal clock (CTC)
- CYC: Cycle-accurate Timer

Intel-PT Example



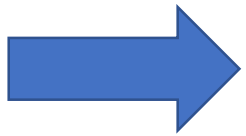
Intel-PT Example



Intel-PT Example

Instructions
push
mov
add
cmp
je .label
mov
.label
call (edx)

PT Trace
CYC 0x1
T
CYC 0x8
0x407e1d8



Cycle	instruction
0x1	push
0x1	mov
0x1	add
0x1	cmp
0x1	je .label
0x8	Call (407e1d8)

Intel-PT Demo

```
// Check whether the CPU supports Intel-PT
$ grep intel_pt /proc/cpuinfo


// Run program with Intel PT
$ sudo perf record -e intel_pt// ./hello
$ sudo perf record -e intel_pt/mtc_period=0,cyc/ ./hello

// Decode
$ sudo perf report -D > trace.dump
```

Intel-PT

+ Can re-construct the sequence of instructions with fine-grained time information
+ Can be used for fine-grained latency debugger
- Recording overhead 5-20%
- High volume of trace output (~100s MB / CPU / s)
- High decoding overhead

More Information on Intel-PT

```
$ man perf-intel-pt
```

Magic Trace: Another tool to produce / analyze Intel-PT traces (with visualization)
https://github.com/janestreet/magic-trace

Use Cases

- Latency Profiling: NSight[NSDI’22], magic-trace
- Introspecting dead code

Open Questions

- Can you minimize the overhead of decoding PT traces?
- Can you make decoding Intel PT more interactive?
- How Intel-PT can be used?