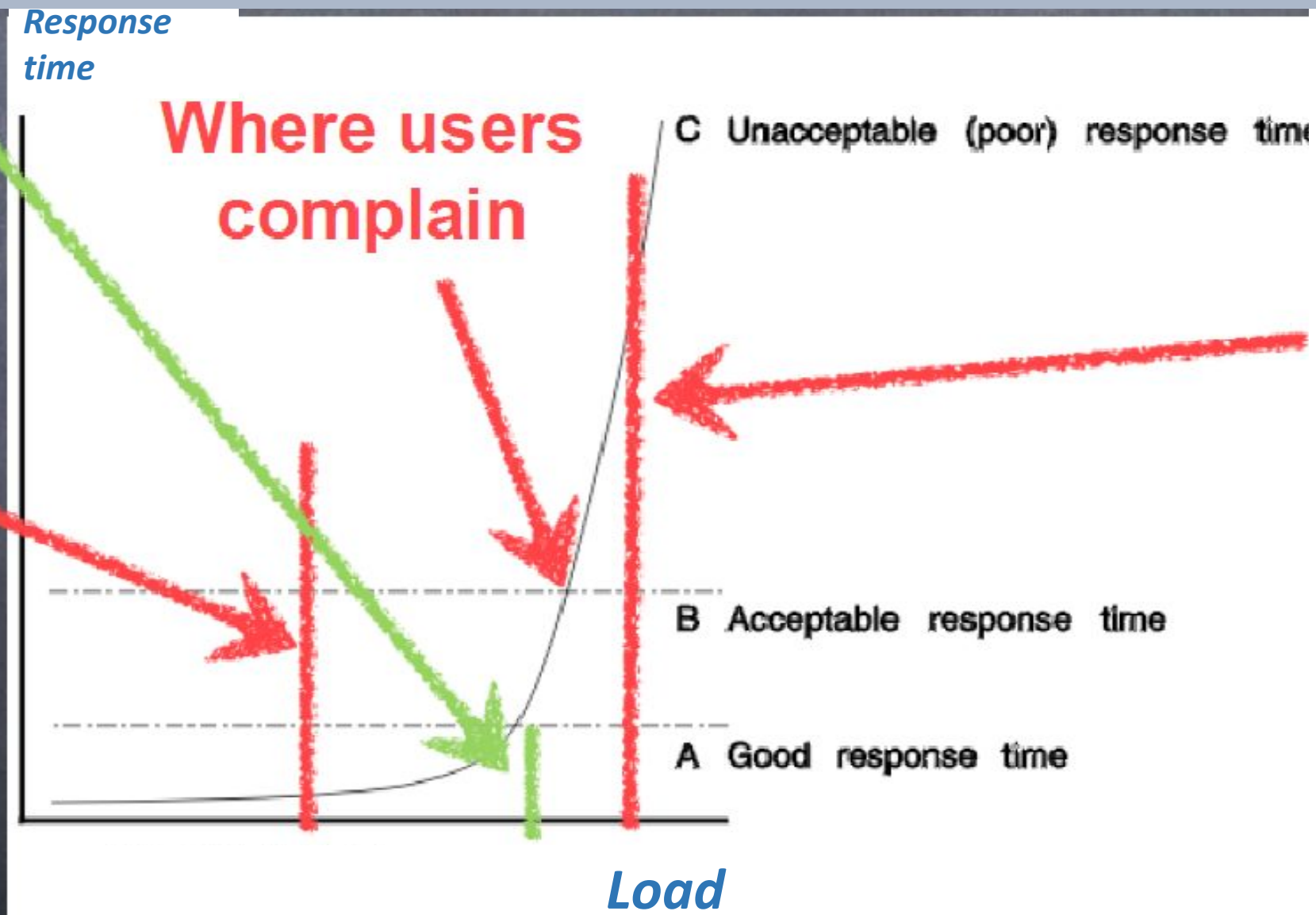
Server Design and Evaluation

Sustainable throughput: The throughput the service can sustain while maintaining safe operation levels

Accurate characterization does matter!

Sustainable Throughput Level

Where the sysadmin is willing to go



Design Goal: High throughput under load

But first need to characterize server performance ….

- What do I measure?
  - Response time (mean, median, 99%-tile, characterize distribution)
  - Throughput / goodput
  - Fairness
  - Resource usage efficiency: e.g., CPU load on server (what's the bottleneck resource?)
- What does workload look like?
  - Identify factors that characterize offered workload
    - request frequency, request inter-arrival time
    - request mix, request characteristics (size)

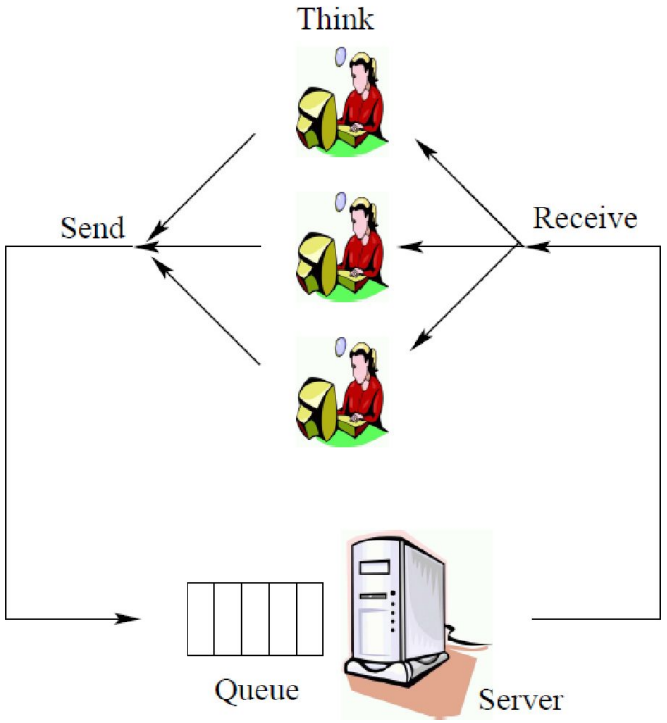
… then decide what to compare against. What does 'performs well' mean?

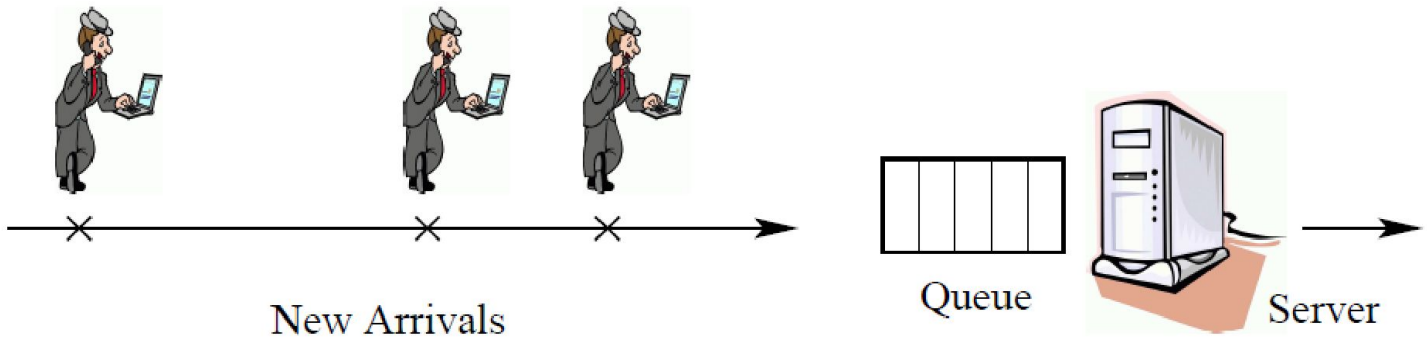
- How would an ideal ('well-conditioned') service perform?

Goal: evaluate the performance of a service

How does one model clients (load generators)?

closed loop vs. open loop





How would an ideal ('well-conditioned') service perform?

(closed-loop generation model for offered load)

- offered load <= capacity →
  - goodput = offered load
  - response time: constant as load grows
  - #dropped requests: none

- offered load > capacity →
  - goodput = capacity
  - response time: linear increase with load (assuming queues can hold pending load)
  - #dropped requests: none (assuming queues can hold pending load)

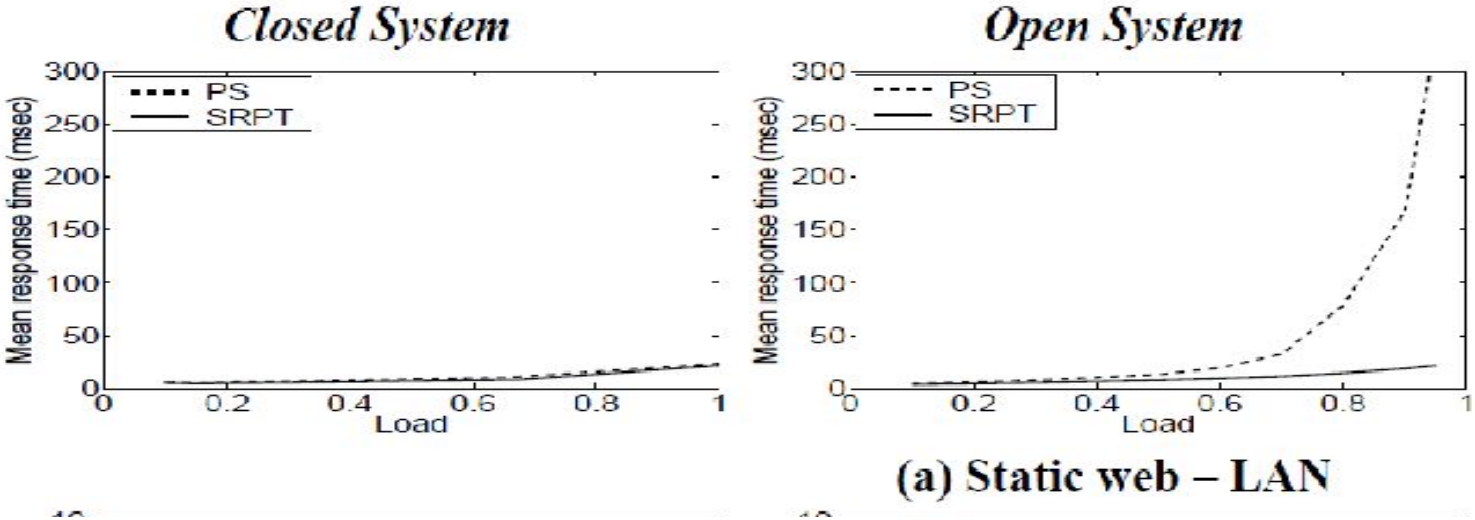
How would an ideal ('well-conditioned') service perform?

(**open-loop** generation model for offered load)

- offered load <= capacity →
  - goodput = offered load
  - response time: constant as load grows
  - #dropped requests: none

- offered load > capacity →
  - goodput = capacity
  - ~~response time: linear increase with load (assuming queues can hold pending load)~~
  - ~~#dropped requests: none (assuming queues can hold pending load)~~

Let’s see some real numbers



(a) Static web – LAN

The workload generation model matters!

(same server load, same throughput)

(50 clients for closed loop system, load adjusted to match load in open system by tinkering with 'think time' in closed one)

Source: "Open vs closed: a cautionary tale.", NSDI'06. pdf

Scheduling policies

FCFS (First-Come-First-Served) Jobs are processed in the same order as they arrive.

PS (Processor-Sharing) The server is shared evenly among all jobs in the system.

PESJF (Preemptive-Expected-Shortest-Job-First) The job with the smallest expected duration (size) is given preemptive priority.

SRPT (Shortest-Remaining-Processing-Time-First): At every moment the request with the smallest remaining processing requirement is given priority.

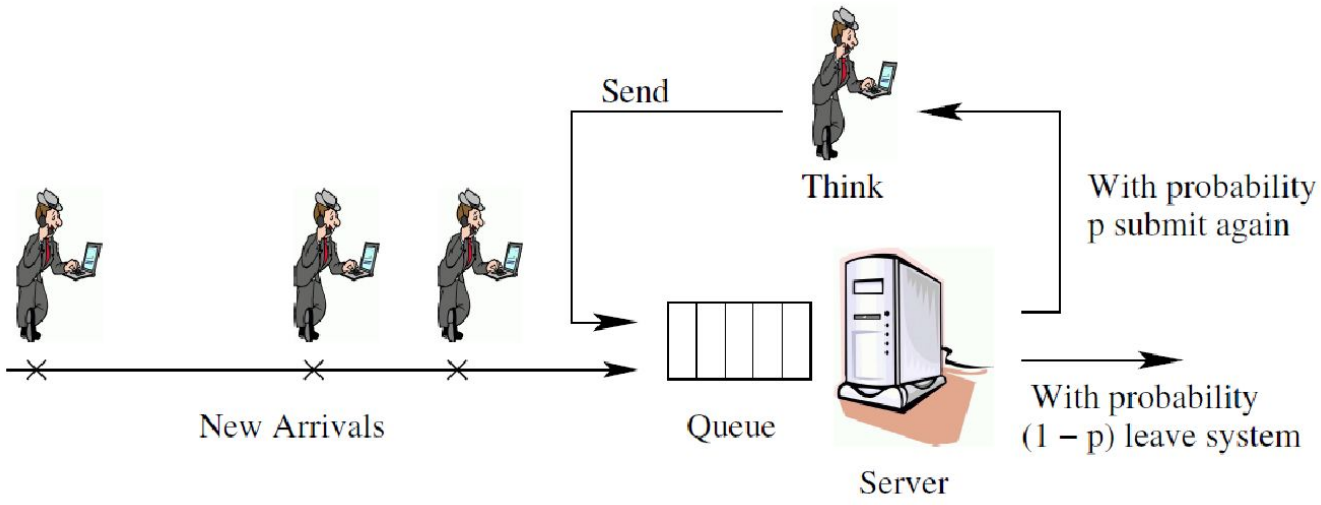
PELJF (Preemptive-Expected-Longest-Job-First) The job with the longest expected size is given preemptive priority. PELJF is an example of a policy that performs badly and is included to understand the full range of possible response times.

The point so far:

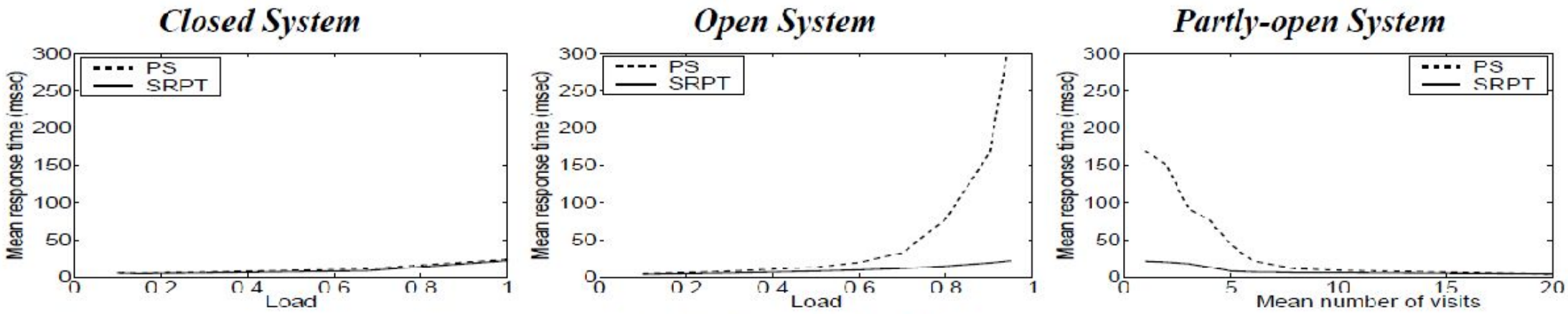
- it is important to accurately characterize response time
- choice of workload generators (open vs. closed) has a large impact
  - [particularly] At high load
  - [particularly] When request size highly variable

However both open- and closed-loop generators are idealized …

A more realistic model: a partly-open system

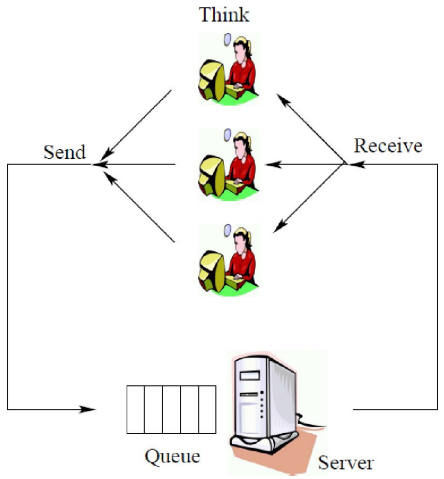


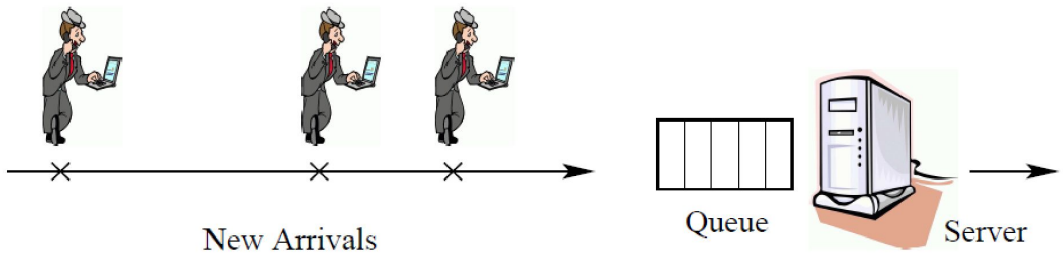
(c) Partly-open system

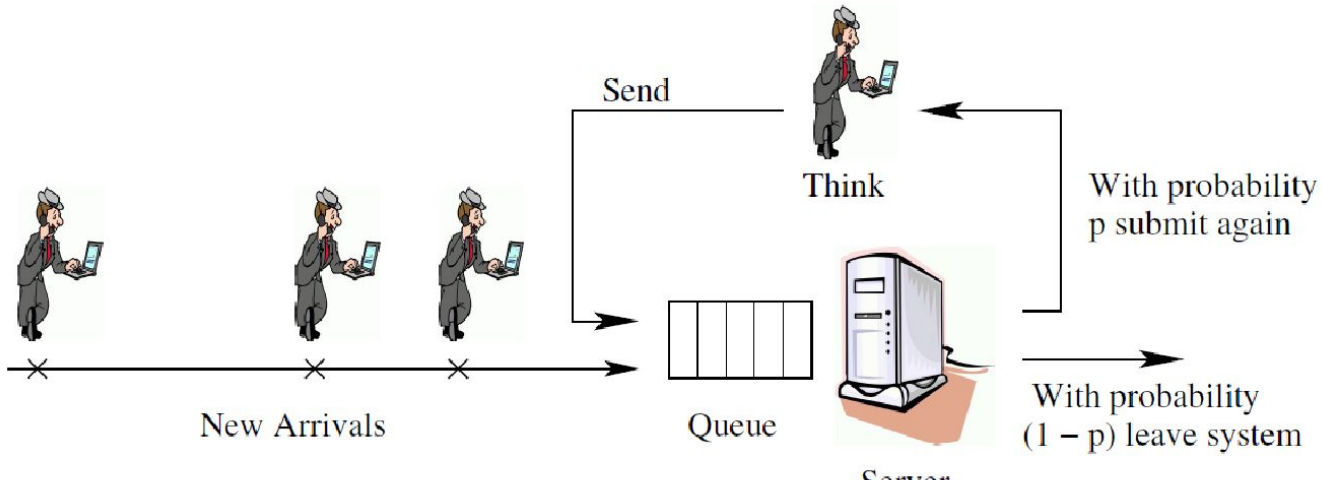


(a) Static web – LAN

Load: 0.9 for partly open system



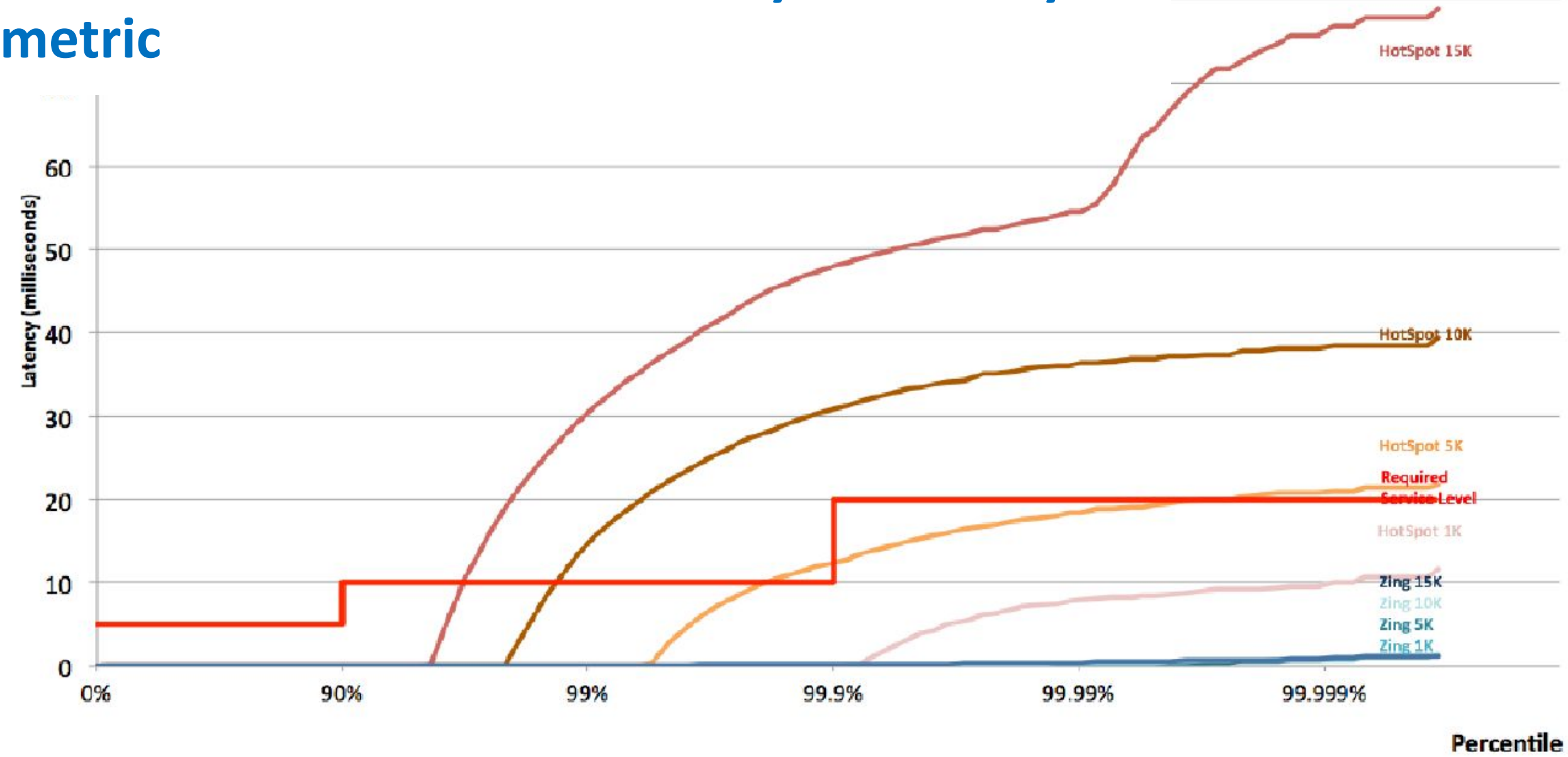




How to chose a model?

- Characterize actual workload
- Determine average number of requests in a ‘session’
- Rules of thumb:
  - >10 req/session □ closed
  - <5 req/session □ open
  - In between: partly open

Focus so far: *mean* response time (RT)
… however RT *distribution* may be the key metric



W3-Q4: Impact of network latency …

Assume

- a remote deployment scenario similar to A4: ~ 10ms latency (one way) between client and server, 128 clients in closed-loop
- a really good implementation of the key/value store.

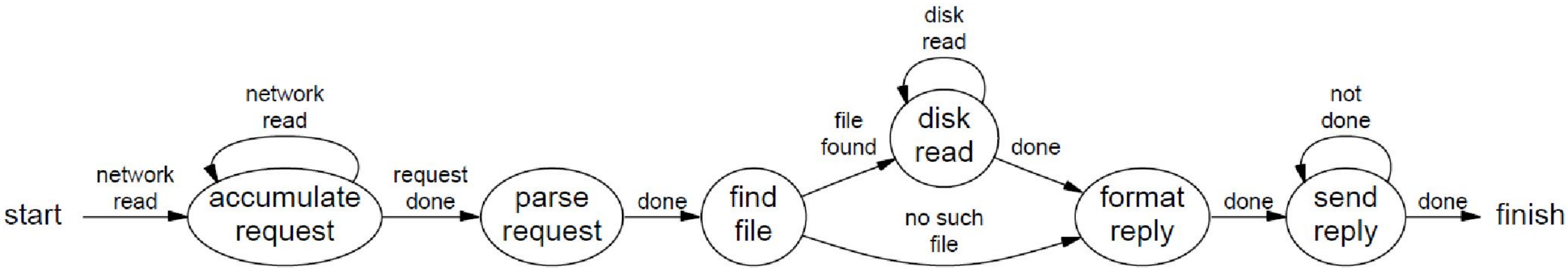
Q: Is there an upper bound for observed throughput?

- No
- Yes: and it is about 1000 req/sec
- Yes: and it is about 6000 req/sec
- Yes: and it is about 50,000 req/sec -- the value I have seen when deploying this on my fancy server at home
- Yes: much higher than any of the above Yes: but lower than any of the above

Recap:

- it is important to accurately characterize response time
- choice of workload generators (open vs. closed) has a large impact
  - [particularly] At high load
  - [particularly] When request size highly variable
- partially-open: alternative to binary choice (open vs. closed)
- response time *distribution* may be important (in addition to mean / median)
- think of upper bounds

Server Design: How may a request look like?



finite-state machine representation for the processing of a simple HTTP request

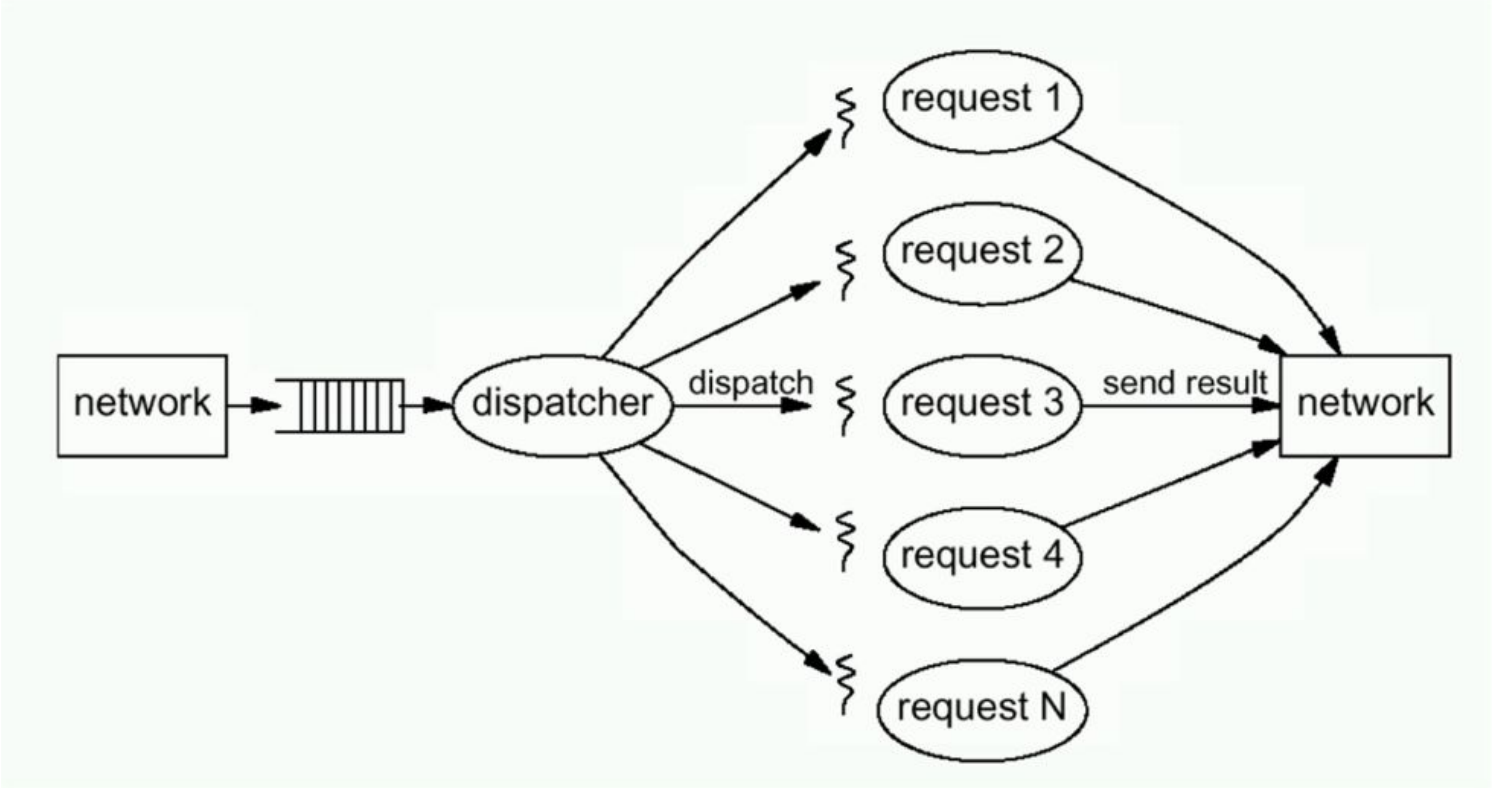
- Can split into stages that have different concerns
- Mix of disk I/O, network, and CPU intensive ‘tasks’.
- Potentially heterogeneous resource requests in the request stream

Server Design: Common Architectural Models

- Thread-based concurrency
- Event-driven server
- Network of queues

Thread-based Concurrency

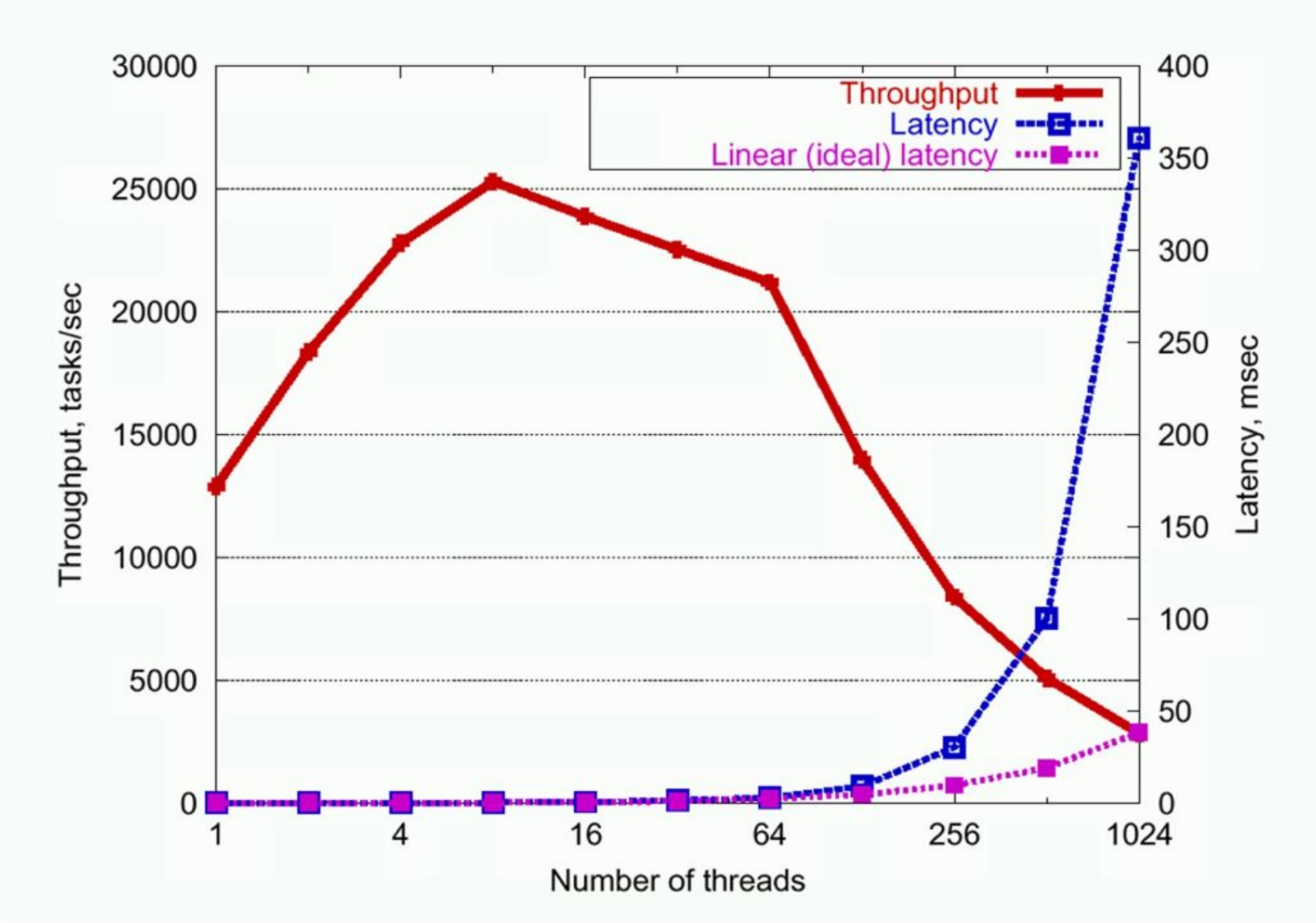
(thread per request allocated/created when receiving the request)



Thread-based Concurrency

- Benefits
  - Simple, well-supported programming model
- Drawbacks
  - High overheads as # of threads increases
    - Cache & TLB misses, scheduling, lock contention

Threaded Server Throughput Degradation



Thread-based Concurrency

- Benefits
  - Simple, well-supported programming model
- Drawbacks
  - High overheads as # of threads increases
    - Cache & TLB misses, scheduling, lock contention

Tension between easiness of programming and ability to introspect and control resources

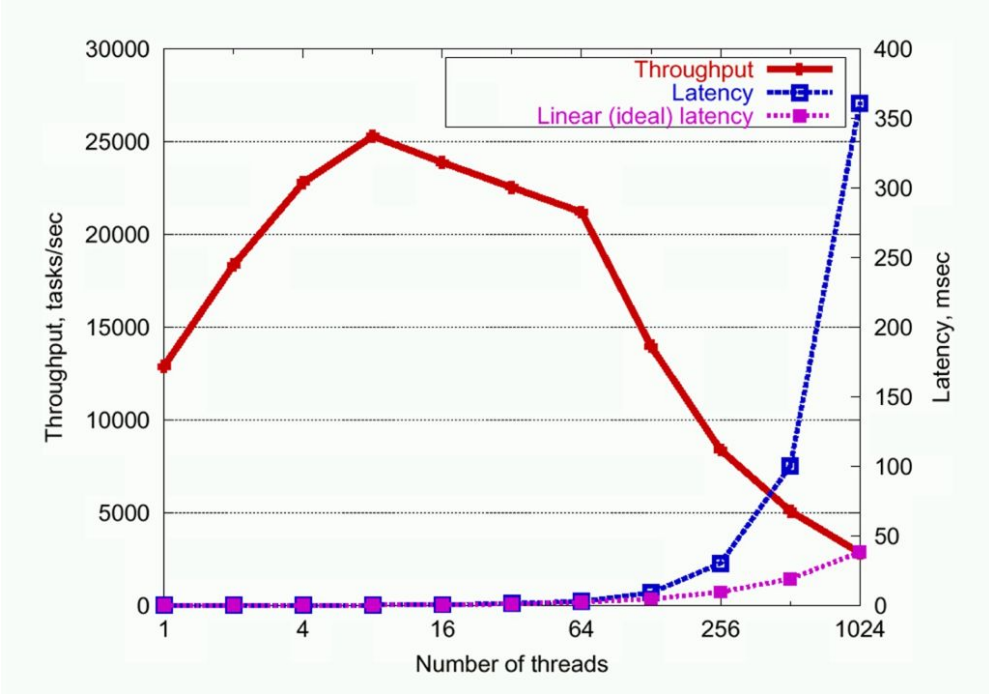
- High cost for the application to characterize resource usage / request (thread)
- Even more complex to arbitrate

… this does not support well a large number of concurrent requests

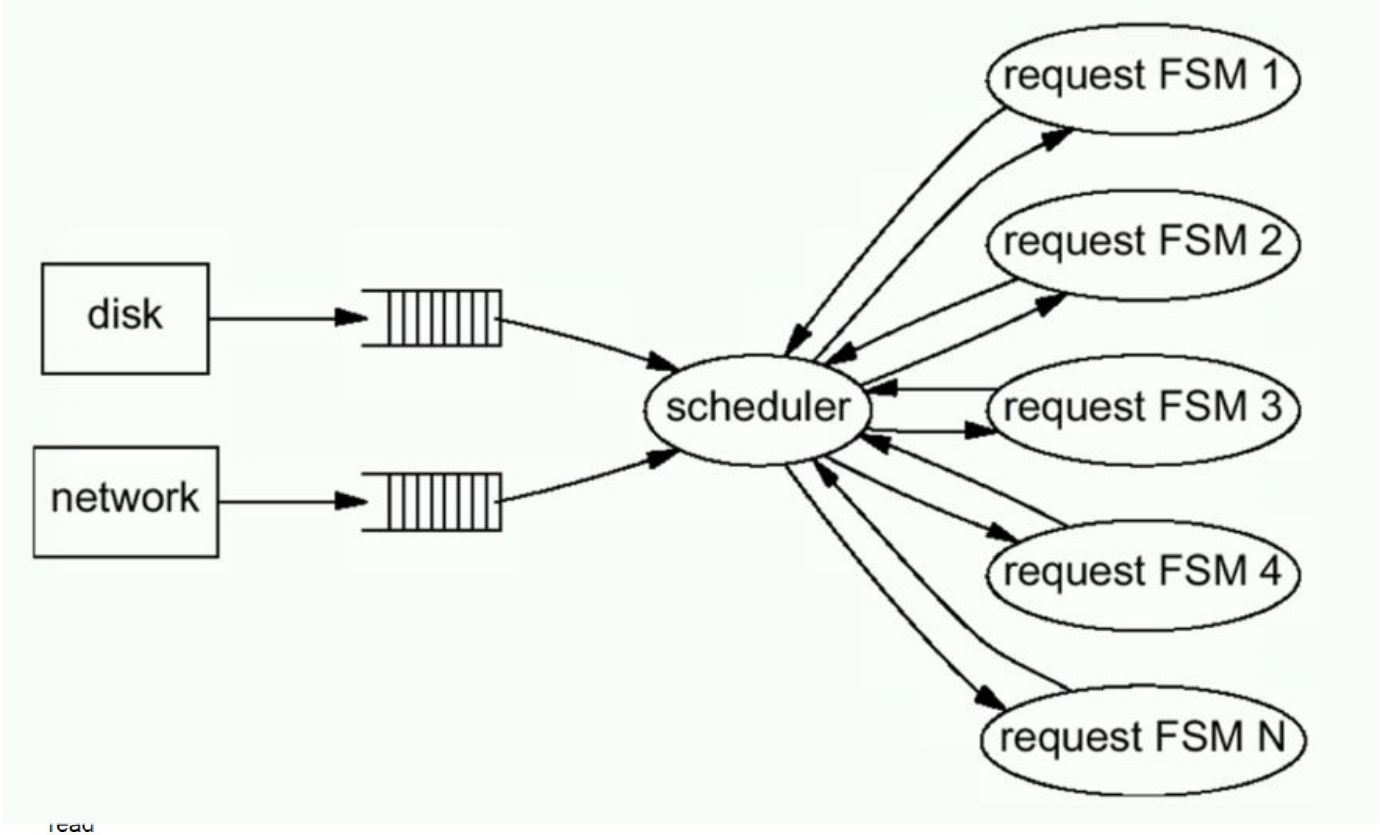
Strawman: let's limit the concurrency level
(bounded thread pool)

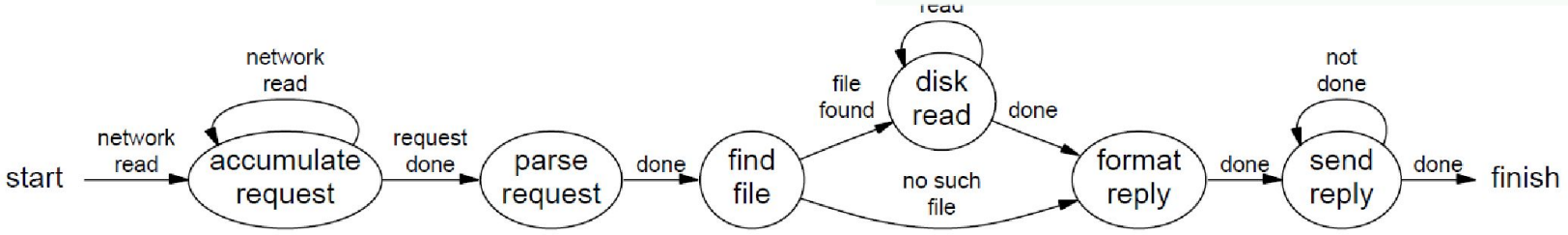
- Does not support my goal of 'well-conditioned' server
  - Unfairness
- Potentially long-running requests hogging all threads

(note though that, depending on your specific environment these may or may not be issues)



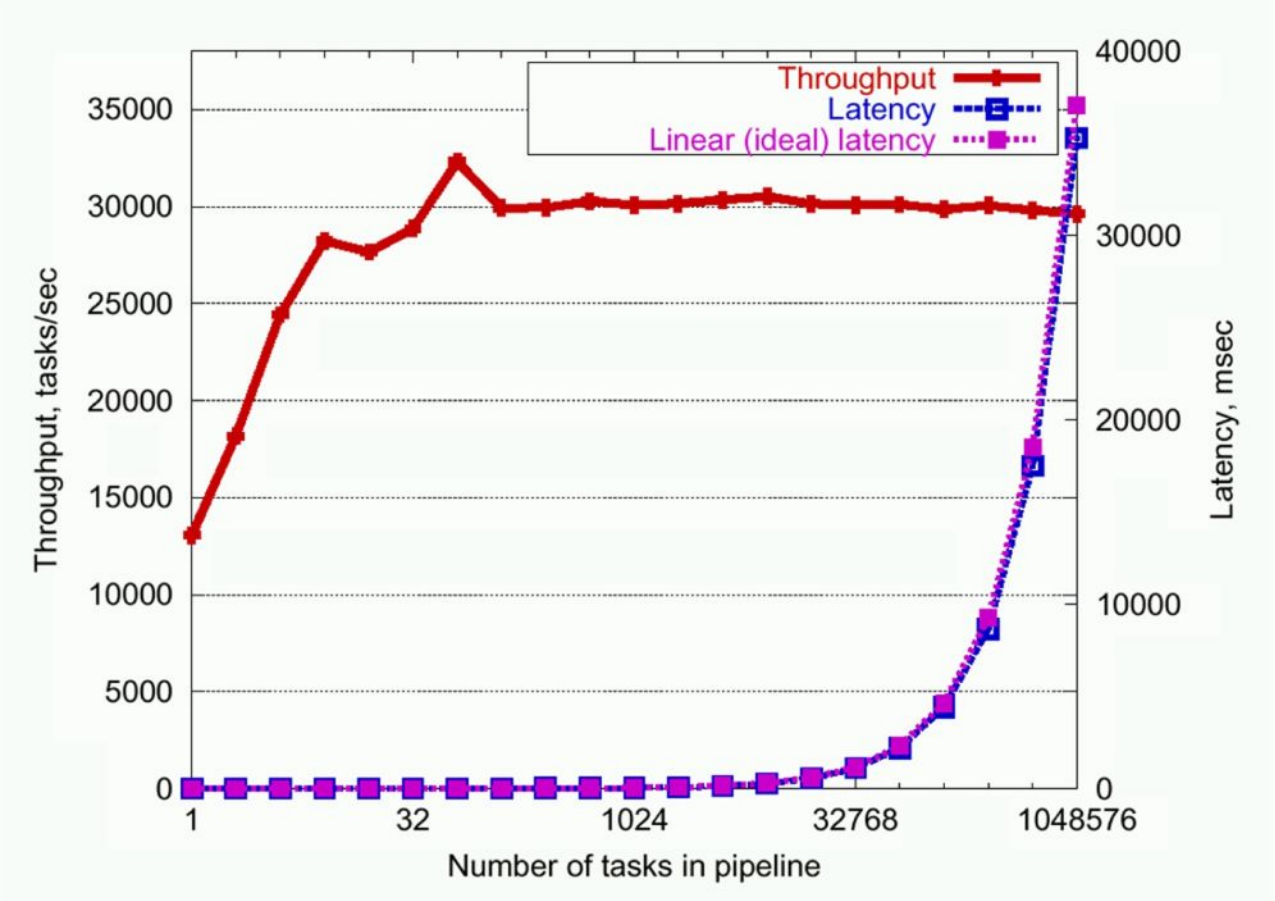
Event-driven Server





One way to think about this: Need to multiplex anyways. State to continue processing the request is maintained by the application (for event-driven) vs. maintained by the OS (thread-based)

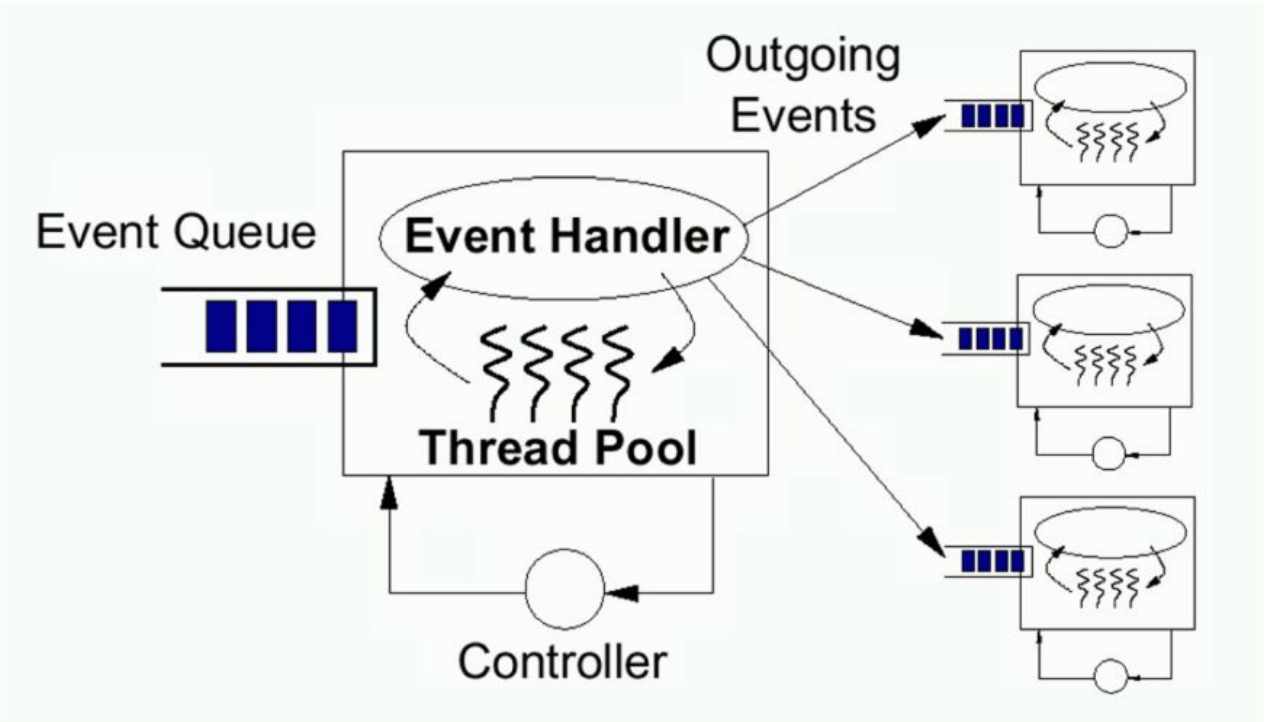
Event-driven Server Throughput



Event-driven Server

- Benefits
  - Robust to high loads
- Drawbacks
  - Harder to program
    - State machine
    - Additional concern on scheduling/ordering events (balance fairness and low response time)
  - Additional overhead if underlying subsystems do not offer an async API.
  - Can not block in event handlers
- More direct control of resources

Network of queues

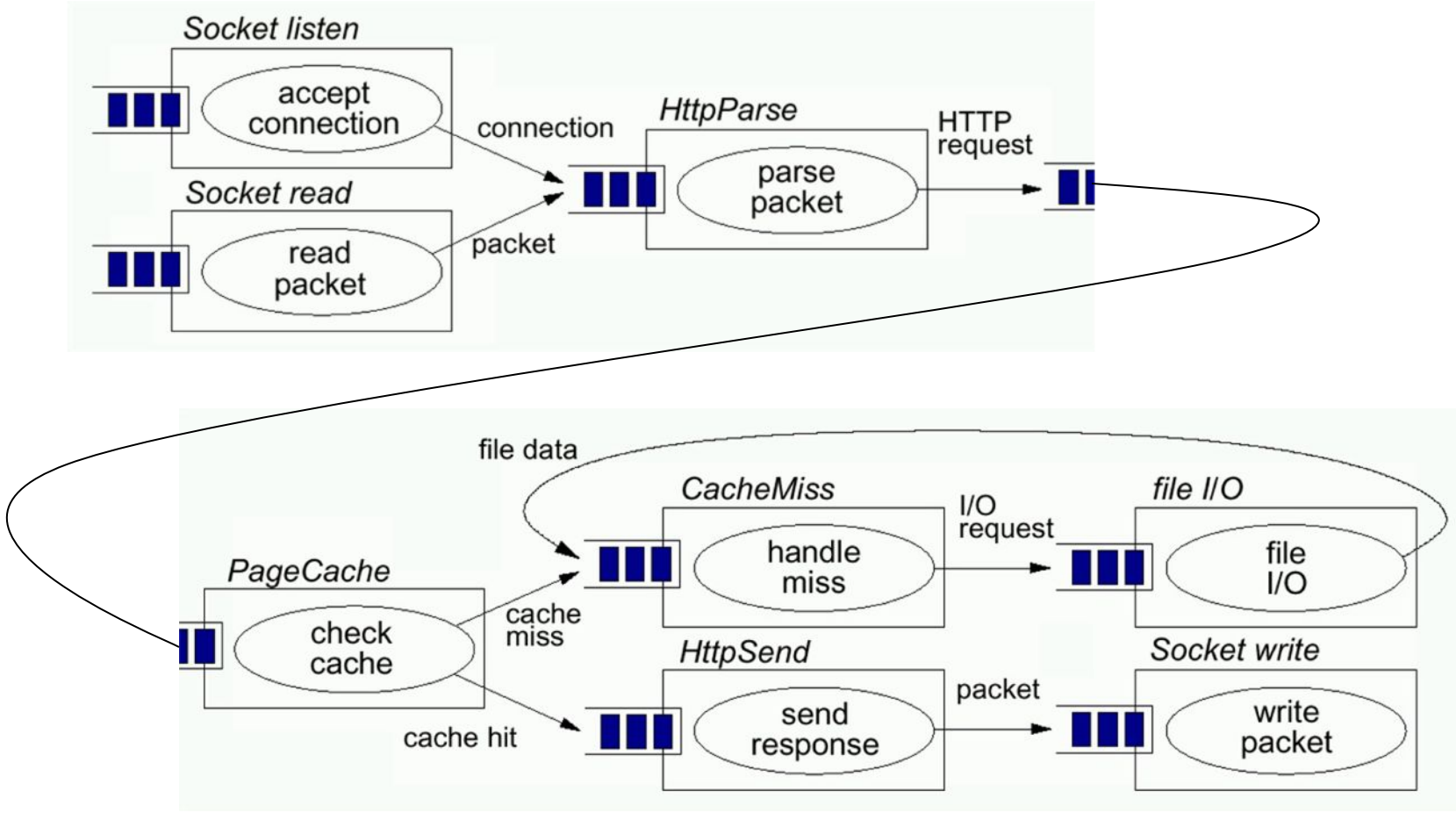


Key mechanisms

- event-driven ***stages***,
- ***resource controllers***

- Support massive concurrency
- Simplify construction of (well-conditioned) services
- Enable introspection
- Support resource management

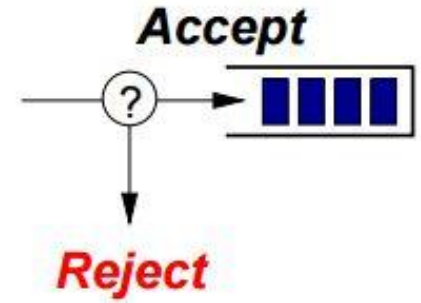
SEDA HTTP Server: Haboob



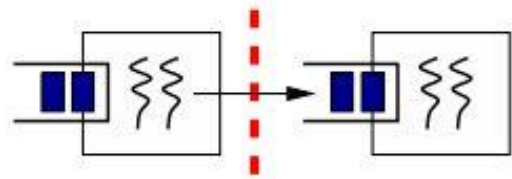
Load and resource bottlenecks explicit in the application programming model.

Queues for Control and Composition

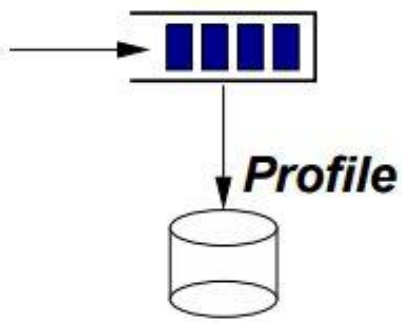
- Queues are *finite*
  - An enqueue behavior may fail
  - Block on full queue -> backpressure
  - Drop rejected events -> load shedding
    - May also do alternative actions, e.g., degraded service



- Queue introduces *explicit execution boundary*
  - Threads may only execute within a single stage
  - Performance isolation, modularity, independent load management



- Explicit event delivery support *inspection*
  - Trace flow of events through application
  - Monitor queue lengths to detect bottleneck



Server Design: Common Architectural Models

- Thread-based concurrency
- Event-driven server
- Network of queues

Different tradeoffs between

- performance, programming effort, fairness, ability to control overload situations, ability to monitor, ability to allocate resources
- optimal choice will also depend on deployment platform and workload characteristics