Architecting for Extreme Overload and Concurrency in Internet Services

Matt Welsh

Harvard University

mdw@eecs.harvard.edu

The Problem: Overload in the Internet

"I tried for three hours to buy tickets online," said Giants fan Alvaro Salinas. "I kept on trying until I started getting errors that forced you out completely, and the chance to buy tickets was no longer available. I know I am not the only one disgusted with their site."

San Francisco Chronicle, Oct. 17, 2002

(The day World Series tickets went on sale.)

Overload happens...

Overload is an inevitable aspect of systems connected to the Internet

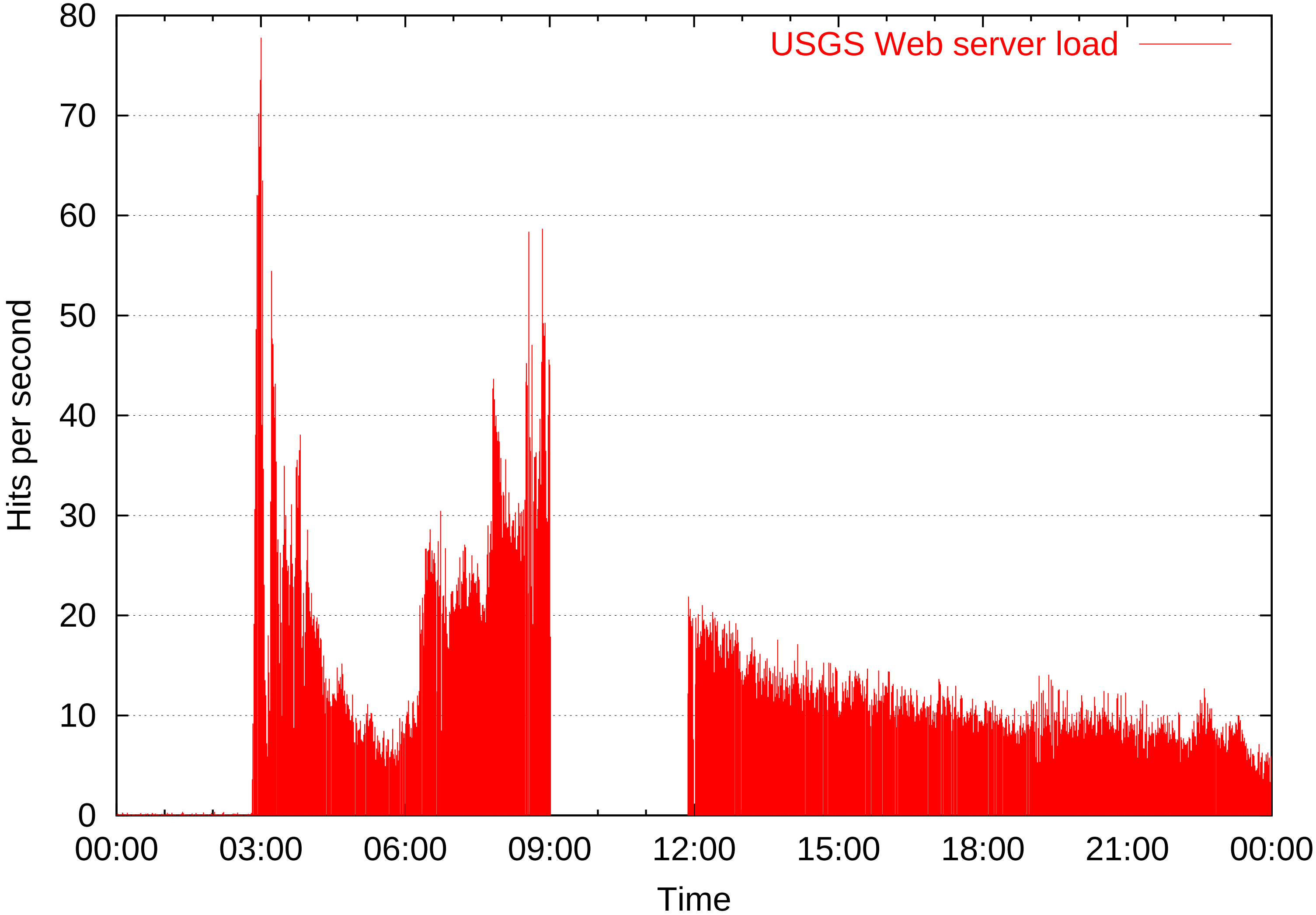
- (Approximately) infinite user populations
- Large correllation of user demand (e.g., flash crowds)
- Peak load can be orders of magnitude greater than average

Some high-profile (and low-profile) examples

- CNN on Sept. 11th: 30,000 hits/sec, down for 2.5 hours
- E*Trade failure to execute trades during overload
- Final Fantasy XI launch in Japan: All servers down for 2 days
- Slashdot effect: daily frustration to nerds everywhere

God’s Version of the Slashdot Effect

Load on USGS Earthquake Information website
following M7.1 earthquake at 3:00 a.m. 10/16/99



Internet Service building is a black art

Supporting massive concurrency is hard!

- Threads don't scale beyond a few hundred
- Throwing more resources at the problem doesn't work
    - ▷ *Can't overprovision when load spikes are 100x or more*

OS doesn't manage heavy load gracefully

- Standard OSs strive for maximum resource transparency
- Setting static resource limits is inflexible
- Load management demands a *feedback loop*

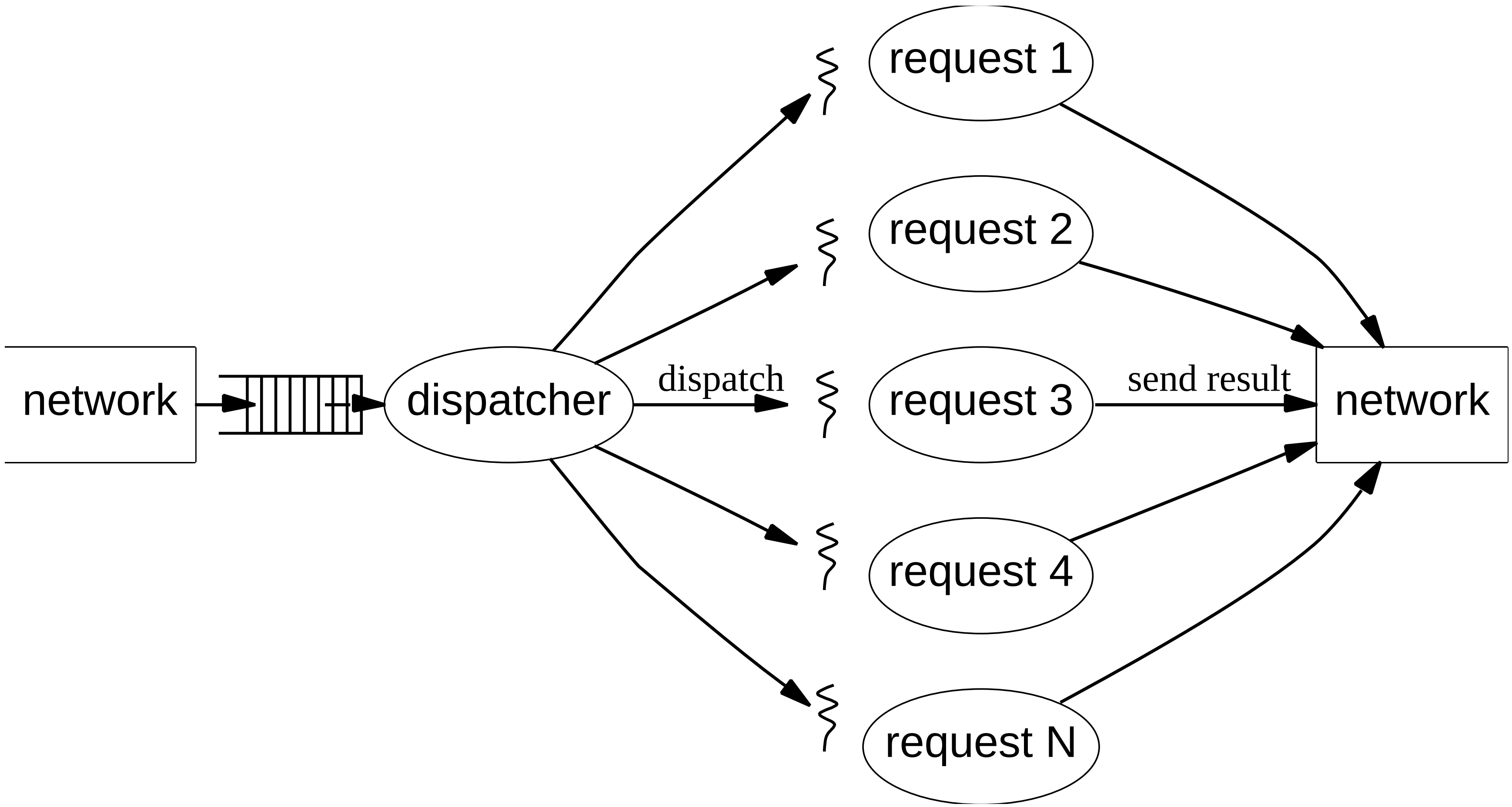
Modern Internet services are extremely dynamic

- Not just about static Web pages anymore
- CGIs, ASPs, Java server pages, database access, SSL
- This makes the load management problem much harder

Outline

- Traditional Internet services - a look under the hood
- Challenges for managing extreme overload
- Our solution: The Staged Event-Driven Architecture
- Adaptive overload control in SEDA
- Lessons learned
- Conclusions

Classic model: One thread per task

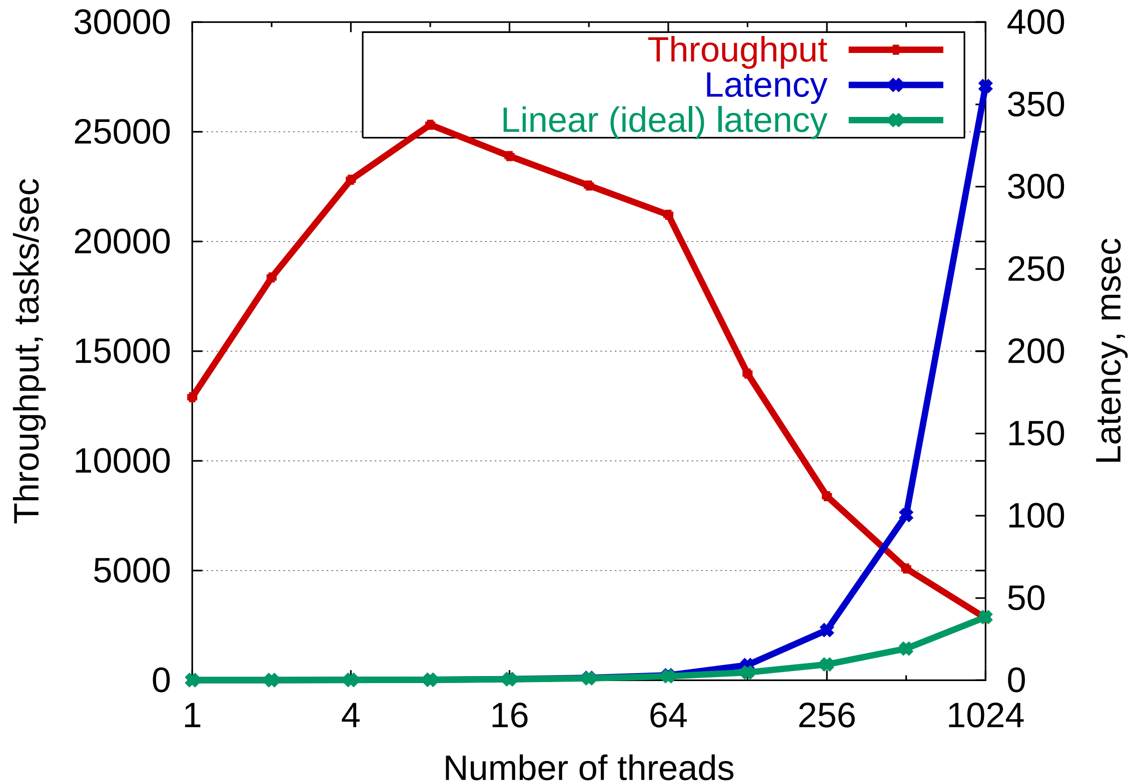


- Create thread per task in system
- Exploit parallelism and I/O concurrency
- Straight-line programming

Problems with Threads

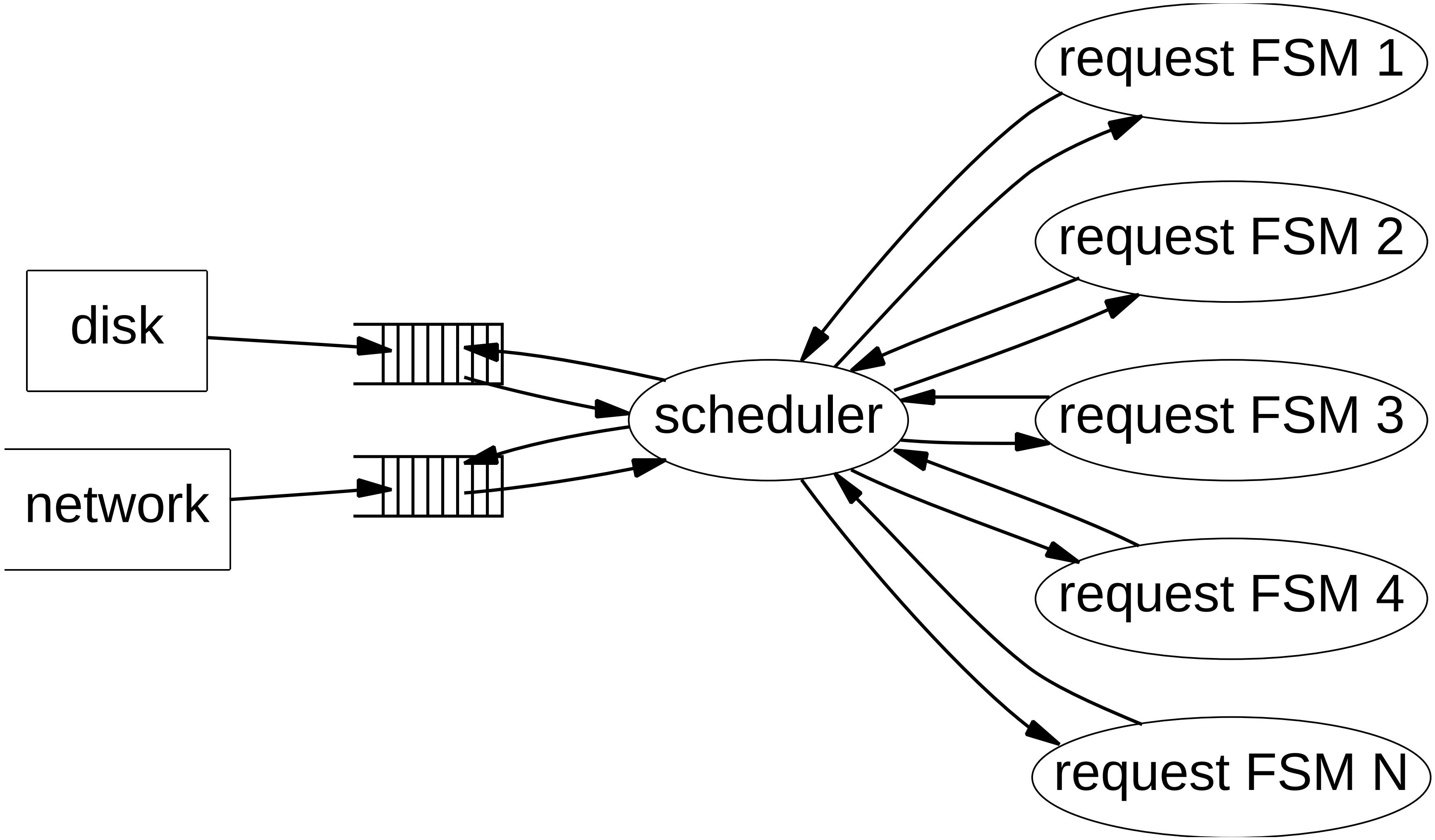
Threads are designed for timesharing

- High resource usage, context switch overhead, contended locks
- Too many threads → throughput meltdown, response time explosion
  - ▷ *How to determine the right number of threads?*
- Regardless of performance, **threads are fundamentally the wrong interface**



(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

Event-based Concurrency



Multiplex many requests (FSMs) over small number of threads

- Single thread processes events
- Each concurrent flow implemented as a finite state machine

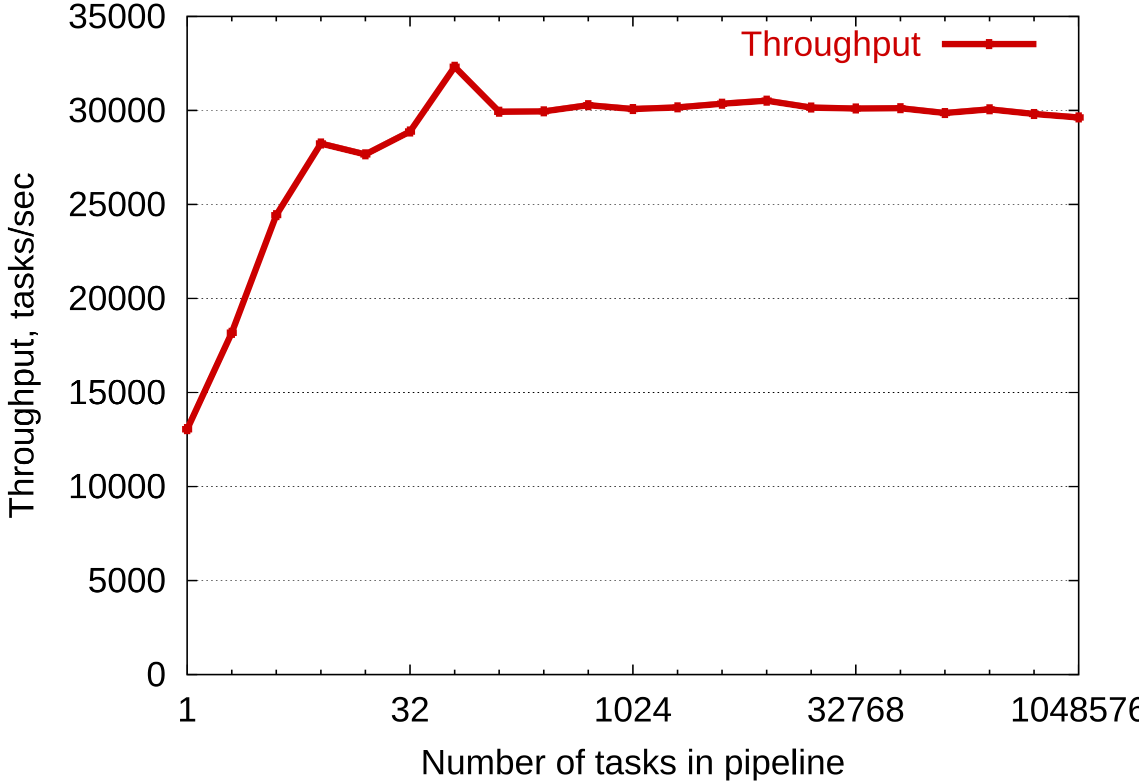
Event-driven Concurrency

Multiplex many requests (FSMs) over small number of threads

- Ideal performance: Flat throughput, linear response time penalty
- Many examples: Click router, Flash web server, TP Monitors, etc.

Difficult to engineer, modularize, and tune

- Typically very brittle: application-specific event scheduling
- FSM code can never block (but page faults, garbage collection force a block)



SEDA: Making Overload Management Explicit

Framework for Internet services that is inherently robust to load

- Scale to large number of simultaneous users/requests
- Degrade gracefully under sudden load spikes
- Address resource management for broad class of Internet services

Design for scalability

- Threads/processes too expensive and cumbersome for concurrency
- Efficient event-driven concurrency coupled with structured design

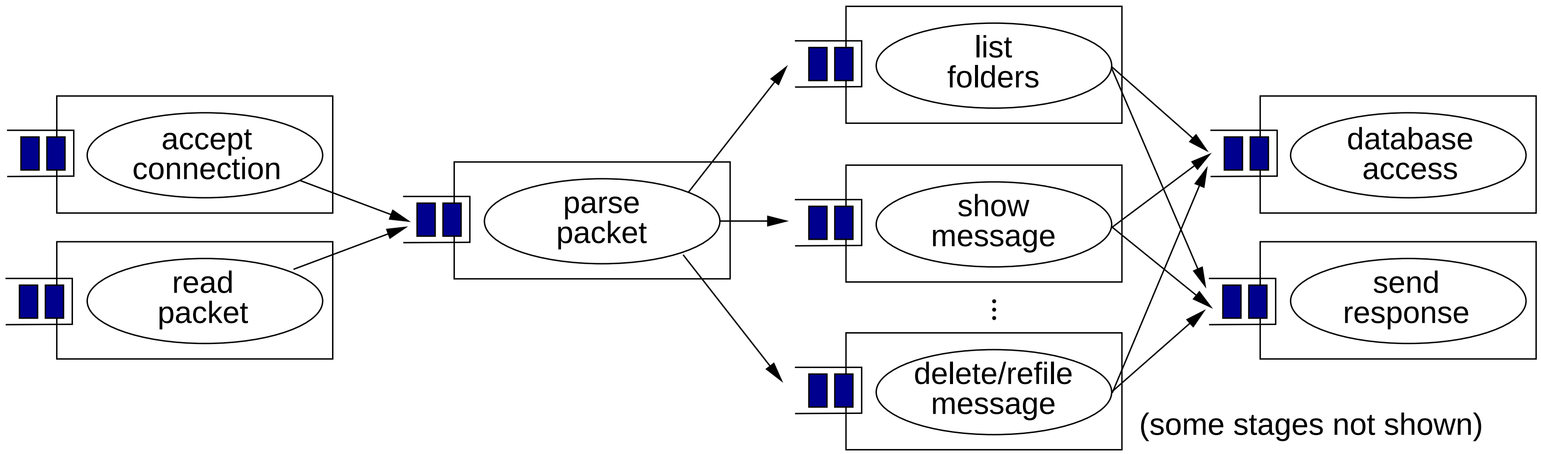
Self-tuning resource management

- System observes performance and adapts resource usage
- Avoid “magic knobs”

Fine-grained admission control

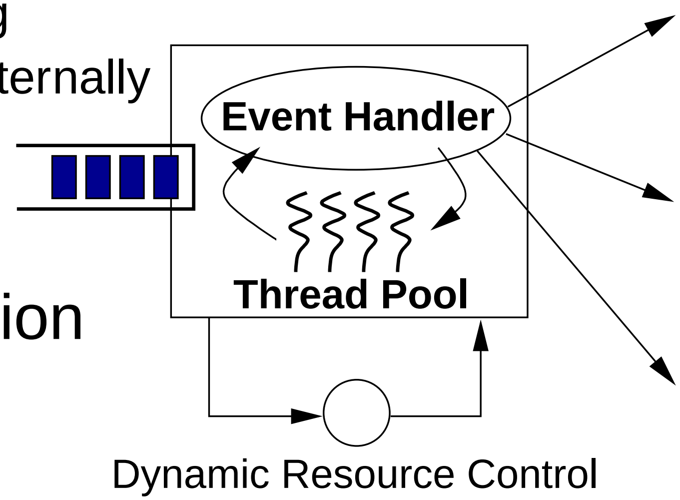
- Control flow of requests **through** service
- Smooth bursts and automatically detect resource bottlenecks

The Staged Event Driven Architecture (SEDA)



Decompose service into *stages* separated by *queues*

- Each stage performs a subset of request processing
- Stages use light-weight event-driven concurrency internally
  - ▷ *Nonblocking I/O interfaces are essential*
- Queues make load management explicit



Stages contain a *thread pool* to drive execution

- Small number of threads per stage
- Dynamically adjust thread pool sizes

Apps don't allocate, schedule, or manage threads

Sandstorm: A SEDA-based Services Platform

An implementation of the SEDA ideas ... in Java

- Augmented Java with nonblocking socket I/O library (NBIO)
- This influenced the design of JDK 1.4 `java.nio` package

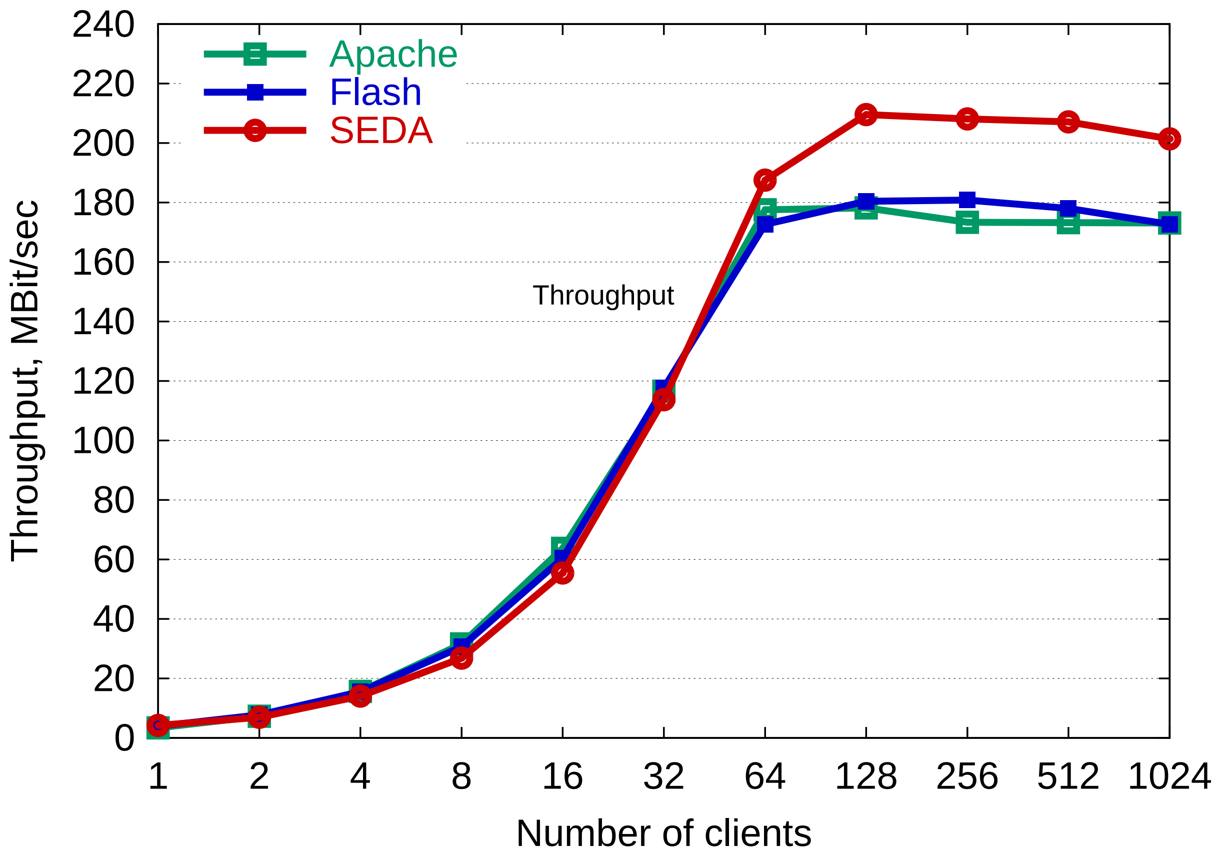
Am I crazy?

- Modern JVMs and JIT compilers perform extremely well
- The benefits of clean design, type safety, and GC outweigh any performance penalty
- Besides, this is about *robust performance*, not *speed*
- Sandstorm's Java-based Web server **outperforms** Apache and Flash!

Source code available at `http://seda.sourceforge.net`

- Incorporated into several commercial products, other research projects

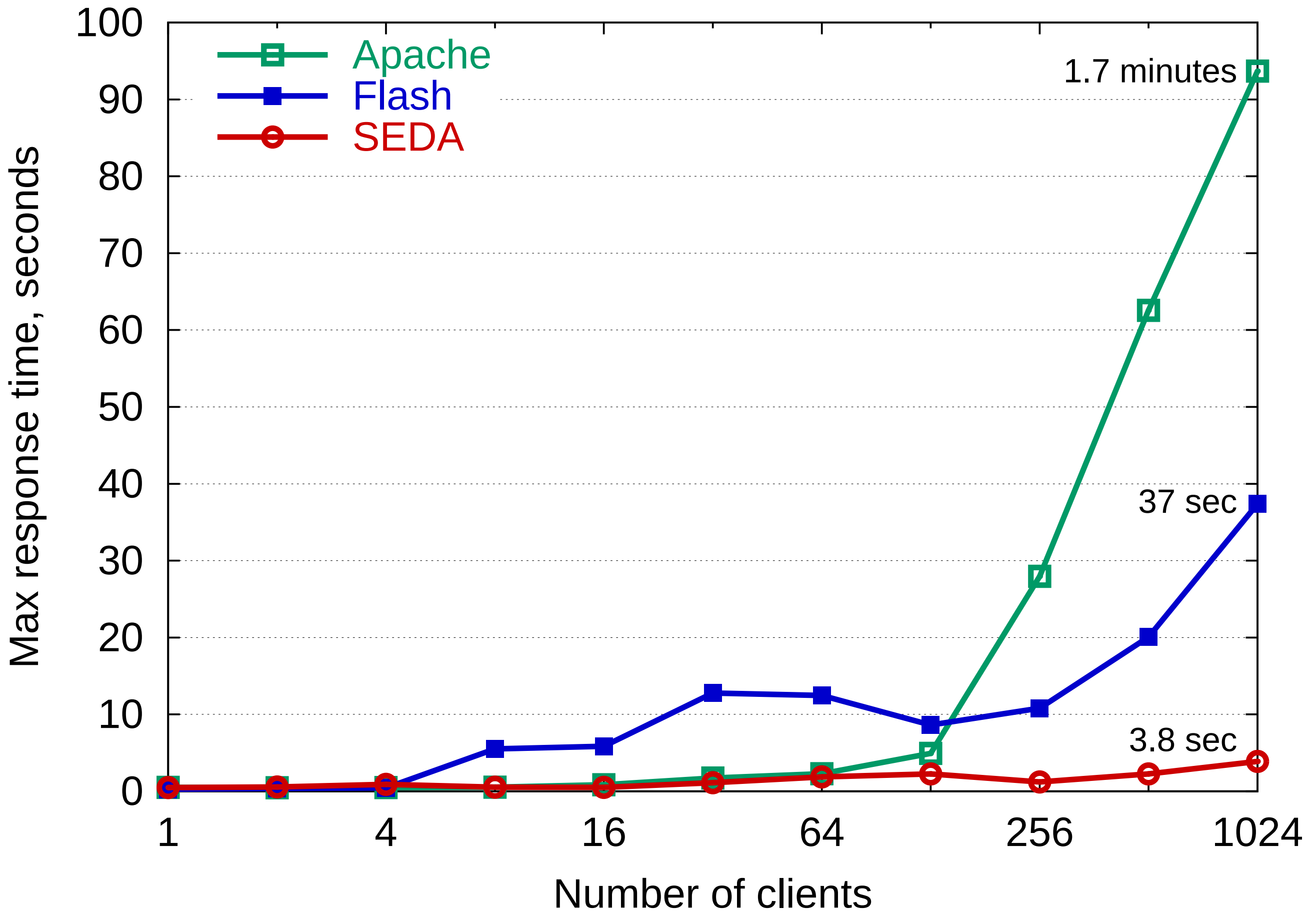
SEDA Scales Well with Increasing Load



4-way Pentium III 500 MHz, Gigabit Ethernet, 2 GB RAM, Linux 2.2.14, IBM JDK 1.3

- SEDA throughput **10% higher** than Apache and Flash (which are in C!)
  - ▷ *But higher efficiency was not really the goal!*
- Apache accepts only 150 clients at once - no overload despite thread model
  - ▷ *But as we will see, this penalizes many clients*

Max Response Time vs. Apache and Flash



- Apache and Flash are very **unfair** when overloaded
  - ▷ *Long response times due to exponential backoff in TCP SYN retransmit*
- Not accepting connections is the **wrong** approach to overload management

Prior work in overload control

Prior work on bounding **system** performance metrics such as:

- CPU utilization, memory, network bandwidth
  - ▷ *No connection to user-perceived performance*
- Instead, we focus on **90th percentile response time**
  - ▷ *Meaningful to users, closely tied to SLAs*

Overload management often based on static resource limits

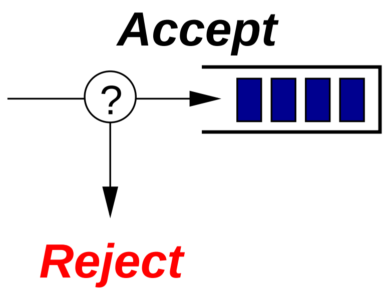
- e.g., Fixed limits on number of clients or CPU utilization
- Can underutilize resources (if limits set too low)
- or lead to oversaturation (if limits too high)

Static page loads or simple performance models

- e.g., Assume linear overhead in size of Web page
- Can't account for dynamic services (scripts, SSL, etc.)

Many techniques studied only under simulation

Exposing overload to applications



Overload is explicit in the programming model

- Every stage is subject to *admission control policy*
- e.g., Thresholding, rate control, credit-based flow control
    - ▷ *Enqueue failure is an* **overload signal**
- Block on full queue → backpressure
- Drop rejected events → load shedding
    - ▷ *Can also degrade service, redirect request, etc.*

```
foreach (request in batch) {
  // Process request...

  try {
    next_stage.enqueue(req);
  } catch (rejectedException e) {
    // Must respond to enqueue failure!
    // e.g., send error, degrade service, etc.
  }
}
```

Alternatives for Overload Control

Basic idea: Apply admission control to each stage

- Expensive stages throttled more aggressively

Reject request (e.g., Error message or "Please wait...")

- Social engineering possible: fake or confusing error message

Redirect request to another server (e.g., HTTP `redirect`)

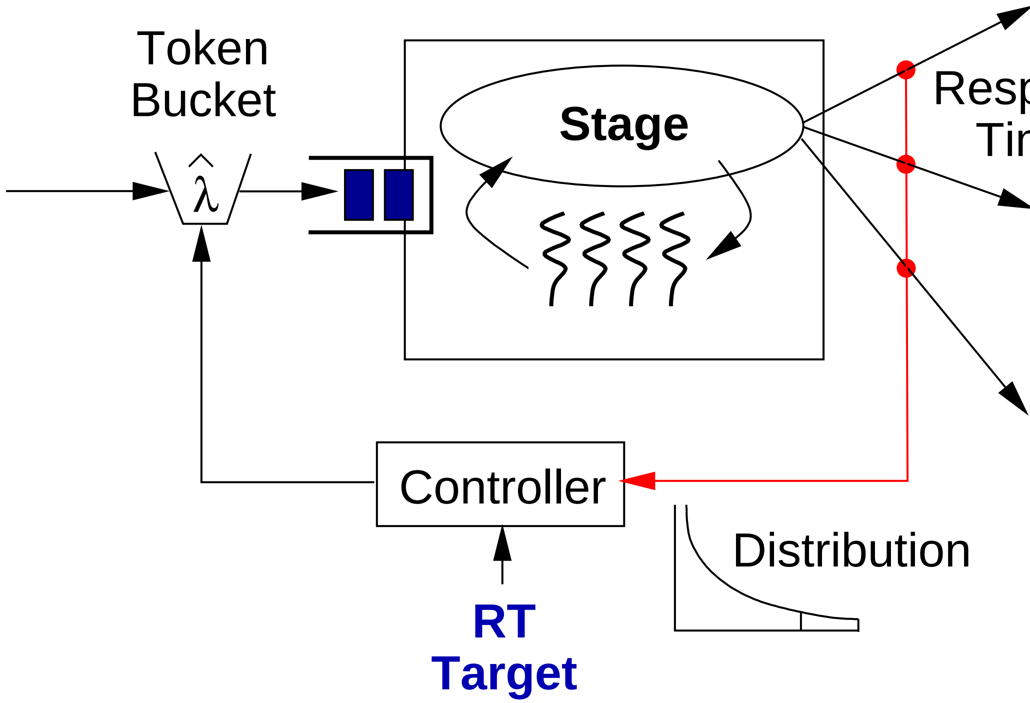
- Can couple with front-end load balancing across server farm

Degrade service (e.g., reduce image quality or service complexity)

Deliver differentiated service

- Give some users better service; don't reject users with a full shopping cart!

Feedback-driven response time control



Adaptive admission control at each stage

- 90th %tile RT target supplied by administrator
- Measure stage latency and throttle incoming event rate to meet target

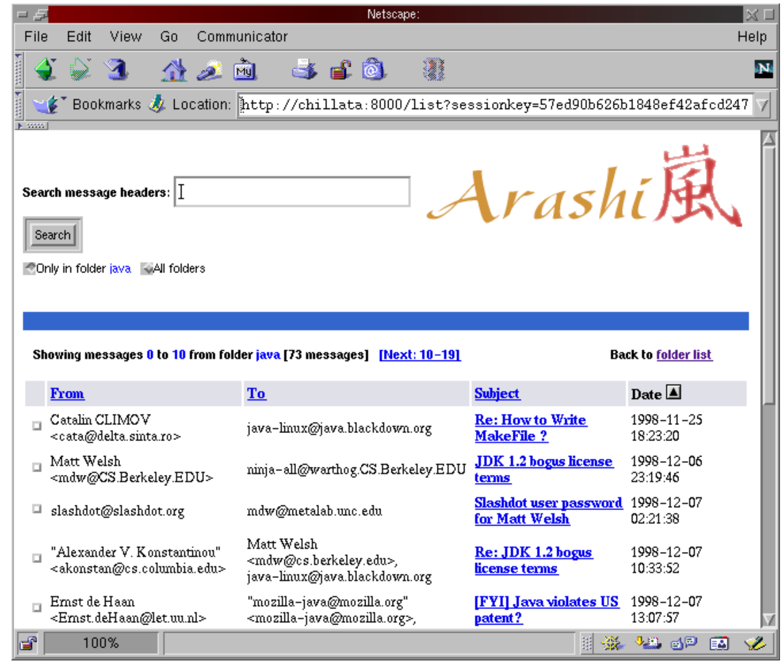
Additive-increase/Multiplicative-decrease controller design

- Slight overshoot in input rate can lead to large response time spikes!
- Clamp down quickly on input rate when over target
- Increase incoming rate slowly when below target

Arashi: A Web-based e-mail service

Yahoo Mail clone - “real world” service

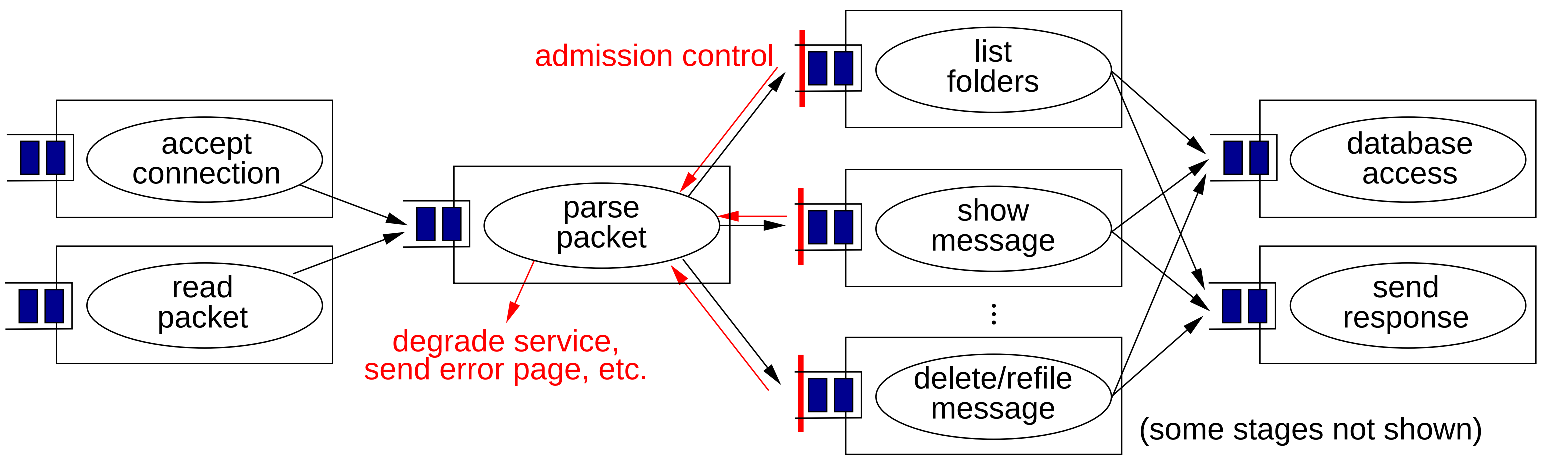
- Dynamic page generation, SSL
- New Python-based Web scripting language
- Mail stored in back-end MySQL database



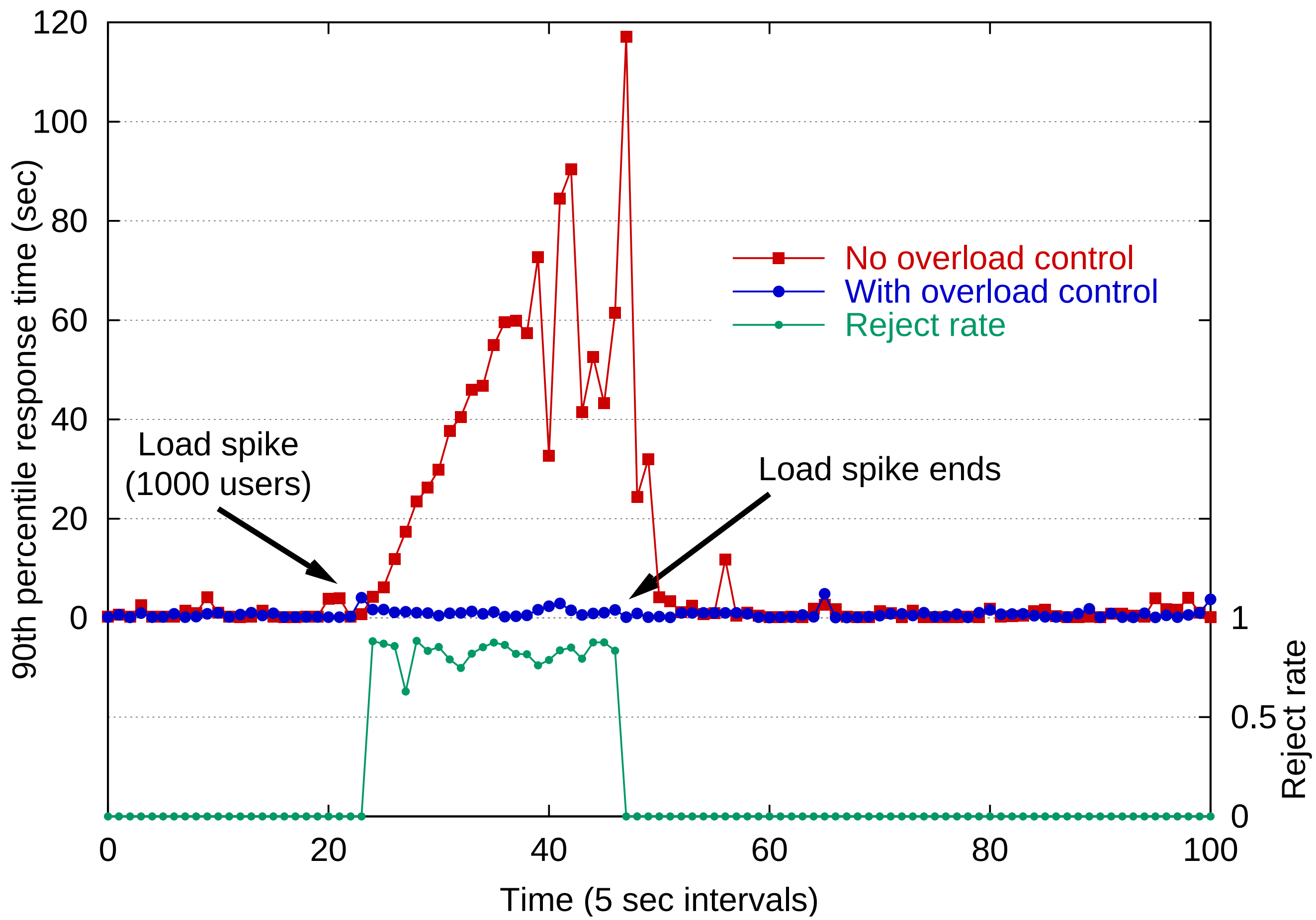
Realistic client load generator

- Traces taken from departmental IMAP server
- Markov chain model of user behavior

Overload control applied to *each request type* separately:



Overload prevention during a load spike

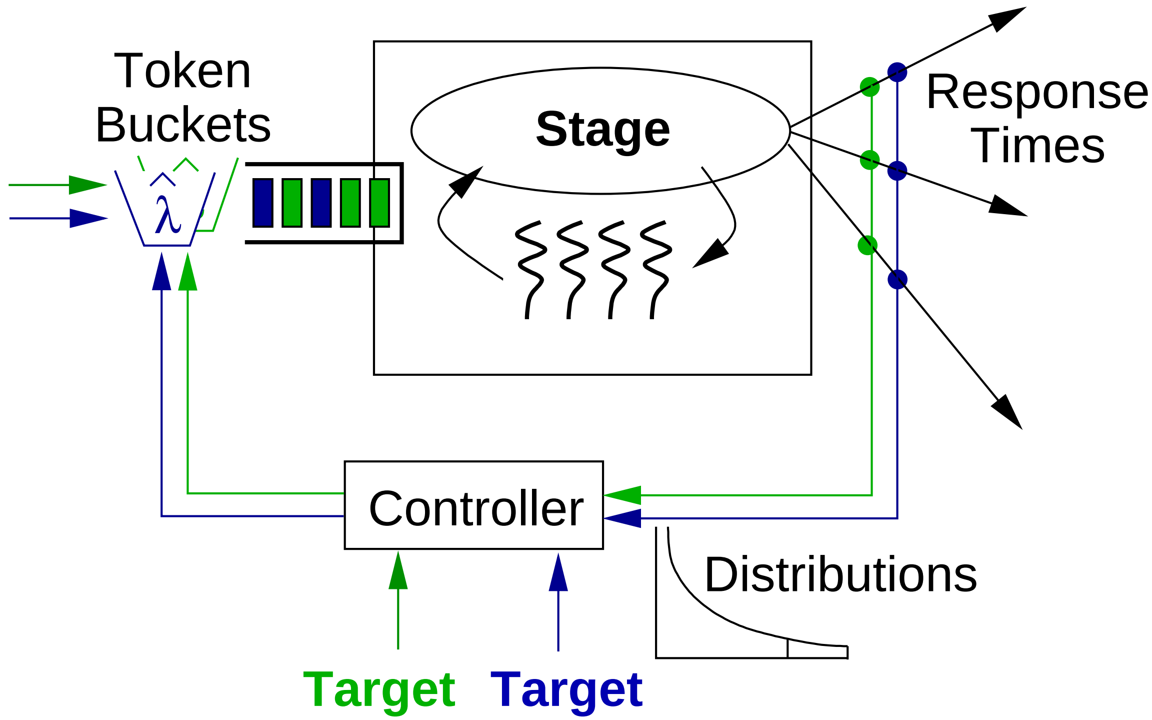


Sudden spike of 1000 users hitting SEDA e-mail service

- 7 request types, handled by separate stages with overload controller
- 90th %tile response time target: **1 second**
- Rejected requests cause clients to pause for 5 sec

Overload controller has no knowledge of the service!

Service Differentiation



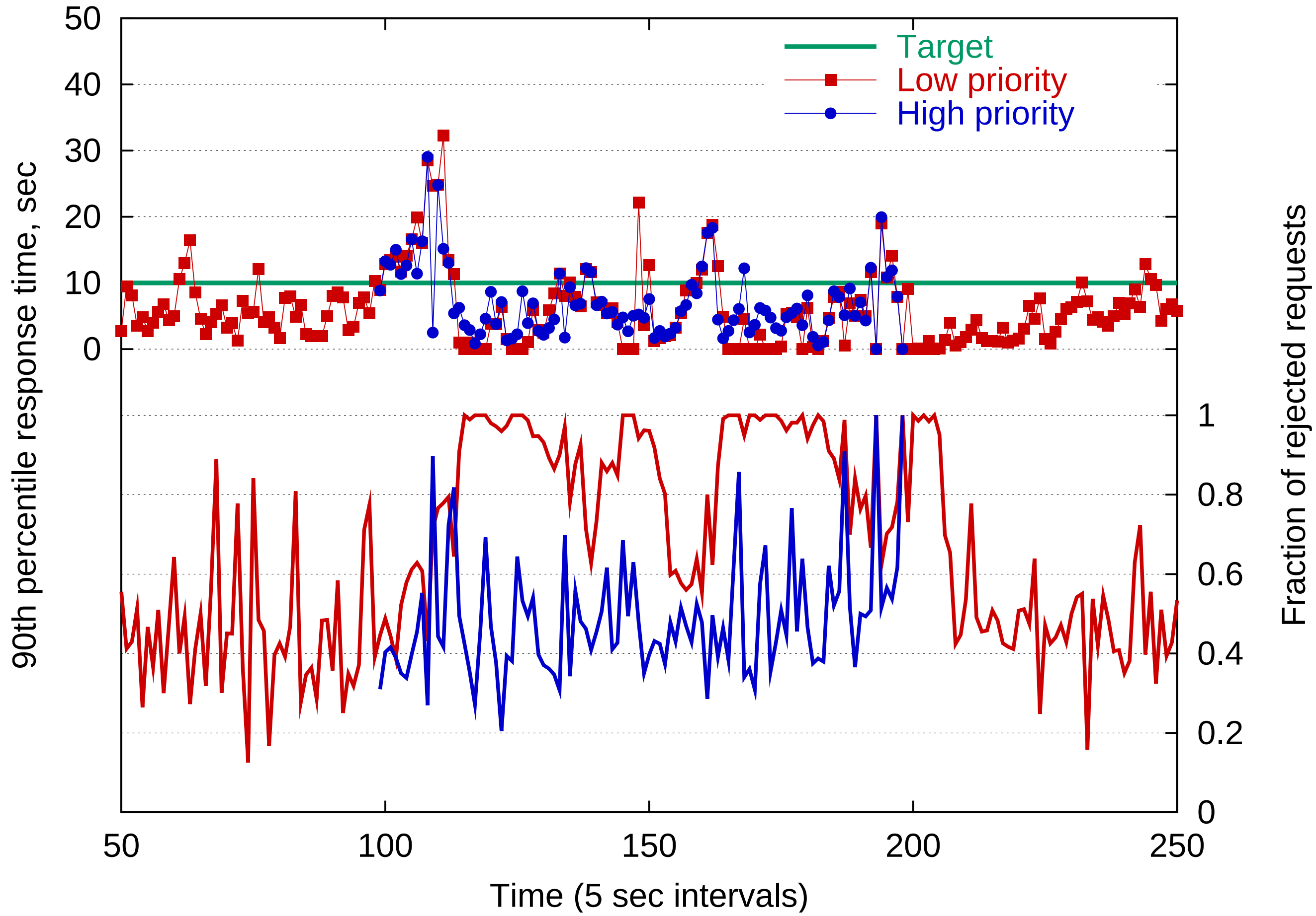
Differentiate users into multiple classes

- Give certain users higher priority than others
- Based on IP address, cookie, header field, etc.

Multiclass admission controller design

- Gather RT distributions for each class, compare to target
  - ▷ *If RT below target, increase rate for* **this class**
  - ▷ *If RT above target, reduce rate of* **lower priority classes**

Service differentiation at work



Two classes of users with a 10 second response time target

- 128 users in each class
- High priority requests suffer fewer rejections
- Without differentiation, both classes treated equally

Why not use control theory?

Increasing amount of theoretical and applied work in this area

- Control theory for physical systems with (sometimes) well-understood behaviors
- Capture model of system behavior under varying load
- Design controllers using standard techniques (e.g., pole placement)
  - ▷ *e.g., PID control of Internet service parameters [Diao, Hellerstein]*
  - ▷ *Feedback-driven scheduling [Stankovic, Abdelzaher, Steere]*

Accurate system models difficult to derive

- Capturing realistic models is difficult
  - ▷ *Highly dependent on test loads*
- Model parameters change over time
  - ▷ *Upgrading hardware, introducing new functionality, bit-rot*

Much control theory based on linear assumptions

- Real software systems highly nonlinear

Playing dodgeball with the kernel

OS resource management abstractions often inadequate

- Resource virtualization hides overload from applications
- e.g., malloc() returns NULL when no memory
- Forces system designers to focus only on "capacity planning"

Internet services require careful control over resource usage

- e.g., Avoid exhausting physical memory to avoid paging
- Back off on processing "heavyweight" requests when saturated

SEDA approach: Application-level monitoring & throttling

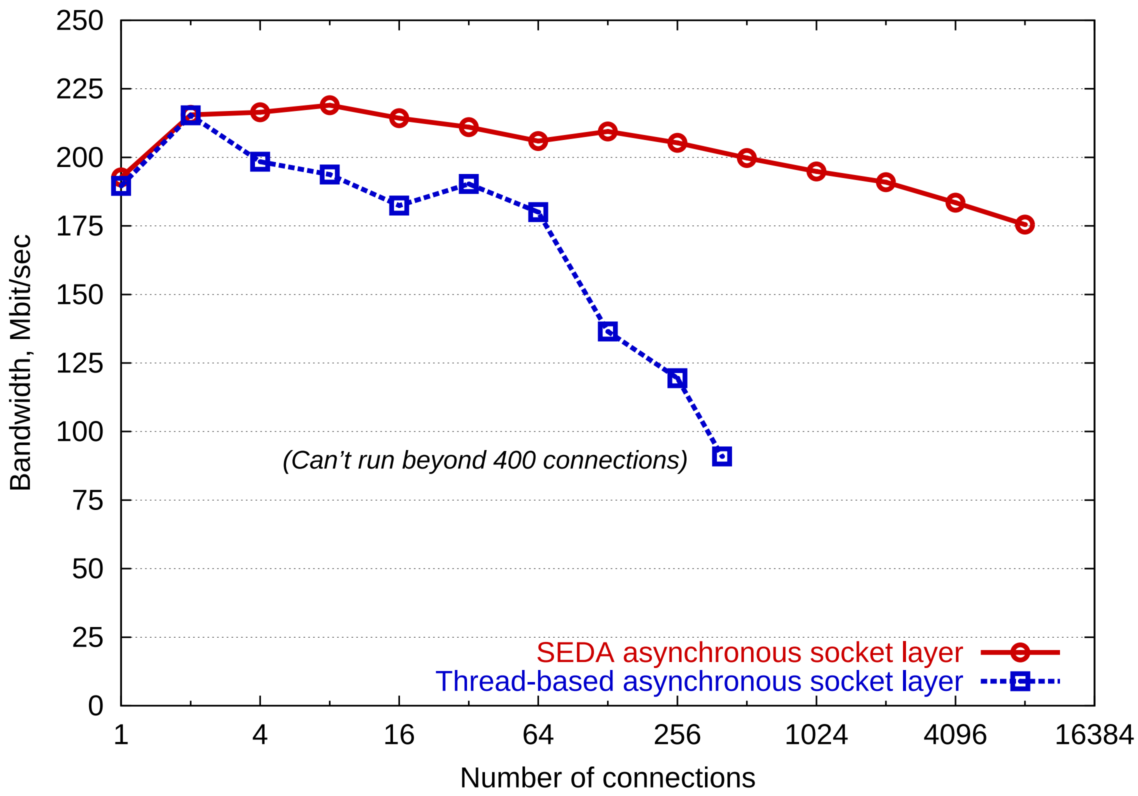
- Service performance monitored at a per-stage level
  - ▷ *Request throughput, service rate, latency distributions*
- Staged model permits careful control over resource consumption
  - ▷ *Throttle number of threads, admission control on each stage*
- Cruder than kernel modifications, but very effective (and clean!)

Scalable concurrency and I/O interfaces

Threads don't scale, but are the wrong interface anyway

- Too coarse-grained: Don't reflect *internal* structure of a service
- Little control over thread behavior (priorities, kill -9)

I/O interfaces typically don't scale



Java as a Service Construction Language

Ease of development was a huge payoff

- Especially when one poor grad student is doing all the coding
- Clean interfaces simplify collaborative development, too
- Performance woes mostly due to garbage collection

JDK 1.4 and the `java.nio` package

- Influenced somewhat on design and experience with NBIO
- `java.nio` more arcane than NBIO
- Performance seems to be comparable

SEDA's aSocket layer (on top of NBIO or `java.nio`)

- Each socket has an outgoing send queue
- Incoming packets arrive on arbitrary stage queue
- Wrinkles: Packets may be processed out of order (by different threads!)

Summary

Focus on *robustness* rather than *raw performance*

- Important to design interfaces that expose resource usage and control
- Don't underestimate value of simple, well-structured framework
- For example: Use of (open) Java rather than (brittle) C/C++

Simple mechanisms often work surprisingly well

- User-level monitoring and control are very effective
- Did not need to resort to complete scheduler and I/O redesign
  - ▷ *But still some juicy problems in that space*

It's all about the interfaces

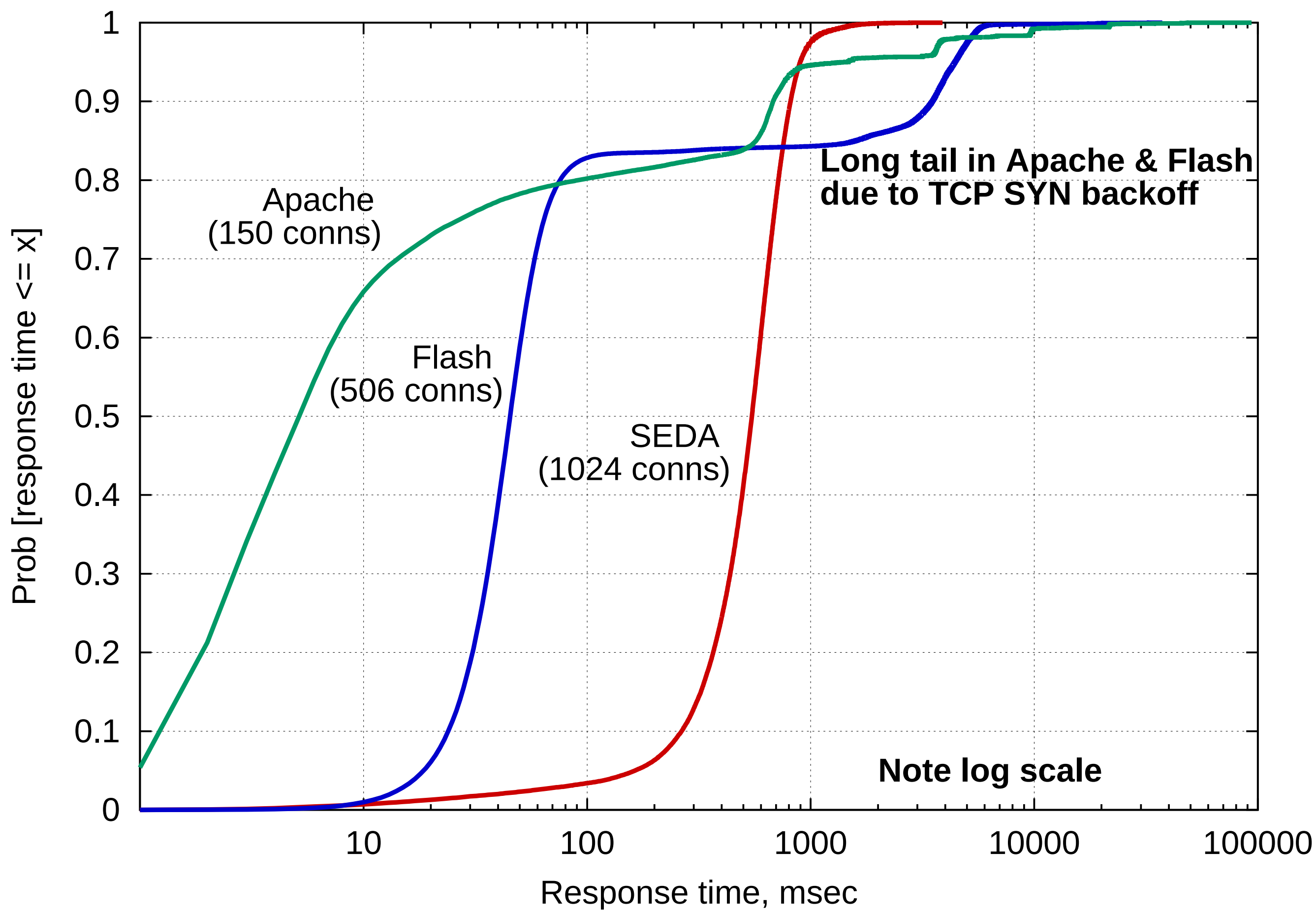
- Applications must have knowledge of (and control over) resource usage
- Common practice is to hide performance tuning in lower layers
- Tradeoff between application complexity and robustness

For more information, software, and papers:

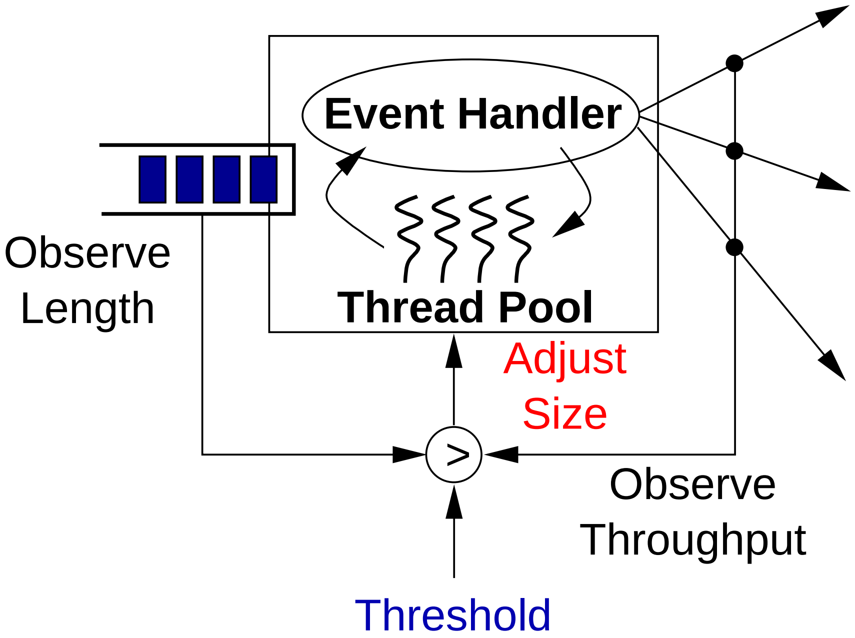
`http://www.eecs.harvard.edu/~mdw/`

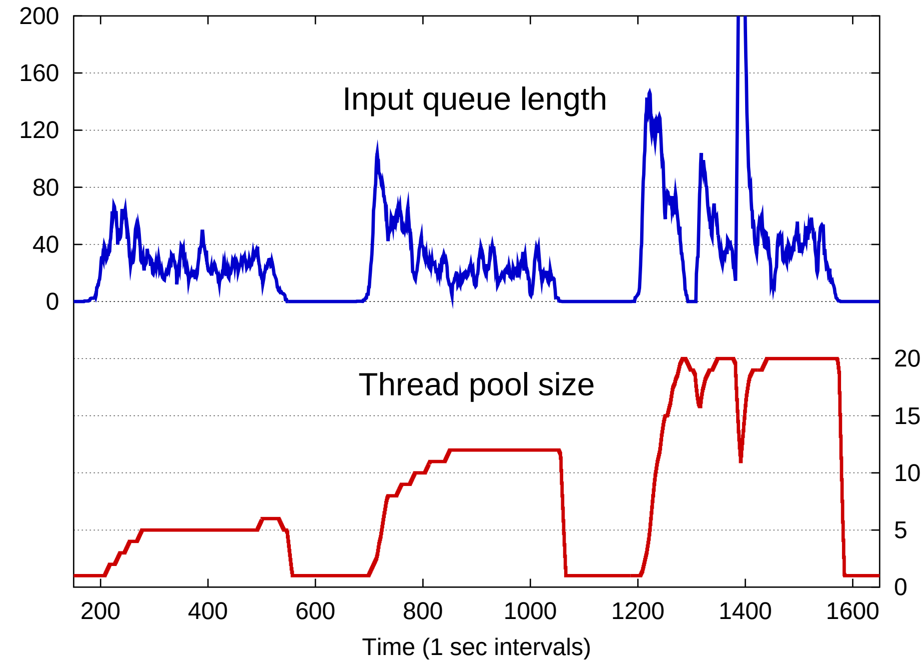
Backup slides follow

Response Time vs. Apache and Flash



SEDA Thread Pool Controller





Goal: *Determine ideal degree of concurrency for a stage*

- Dynamically adjust number of threads allocated to each stage
- Avoid wasting threads when unneeded

Controller operation

- Observes input queue length, adds threads if over threshold
- Idle threads removed from pool
- Drop in stage throughput indicates max thread pool size

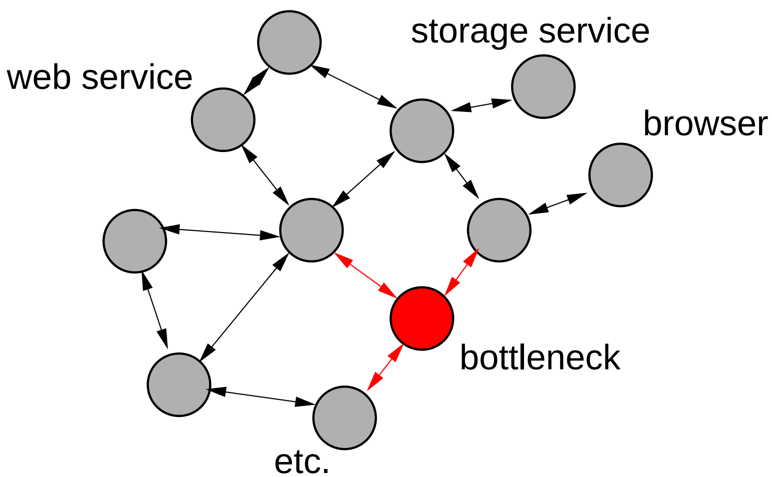
Distributed programming models and protocols

HTTP pushes overload into the network

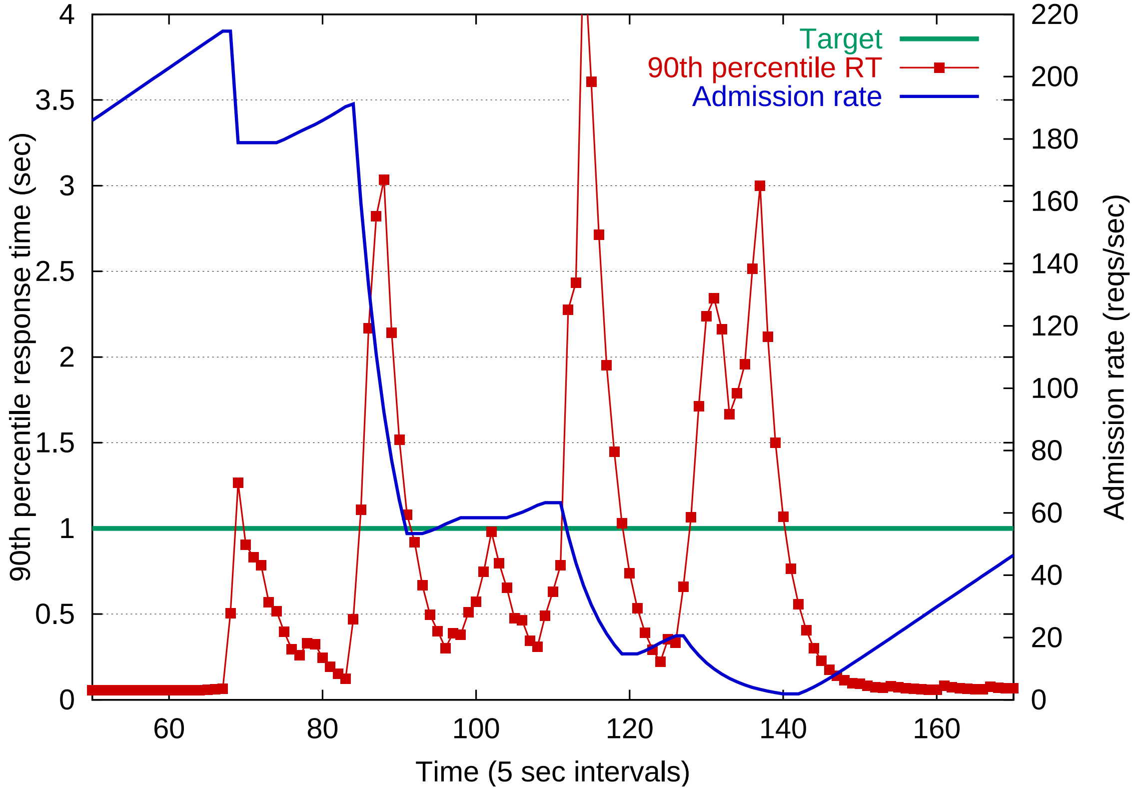
- Relies on TCP connection backoff rather than more explicit mechanisms
- Simultaneous connections, progressive download, and out-of-order requests complicate matters
- Protocol design should consider *service availability*

Distributed computing models generally do not express overload

- CORBA, RPC, RMI, .NET all based on RPC with "generic" error conditions
- On error, should app fail, retry, or invoke an alternate function?
- Single bottleneck in large distributed system causes cascading failure in network



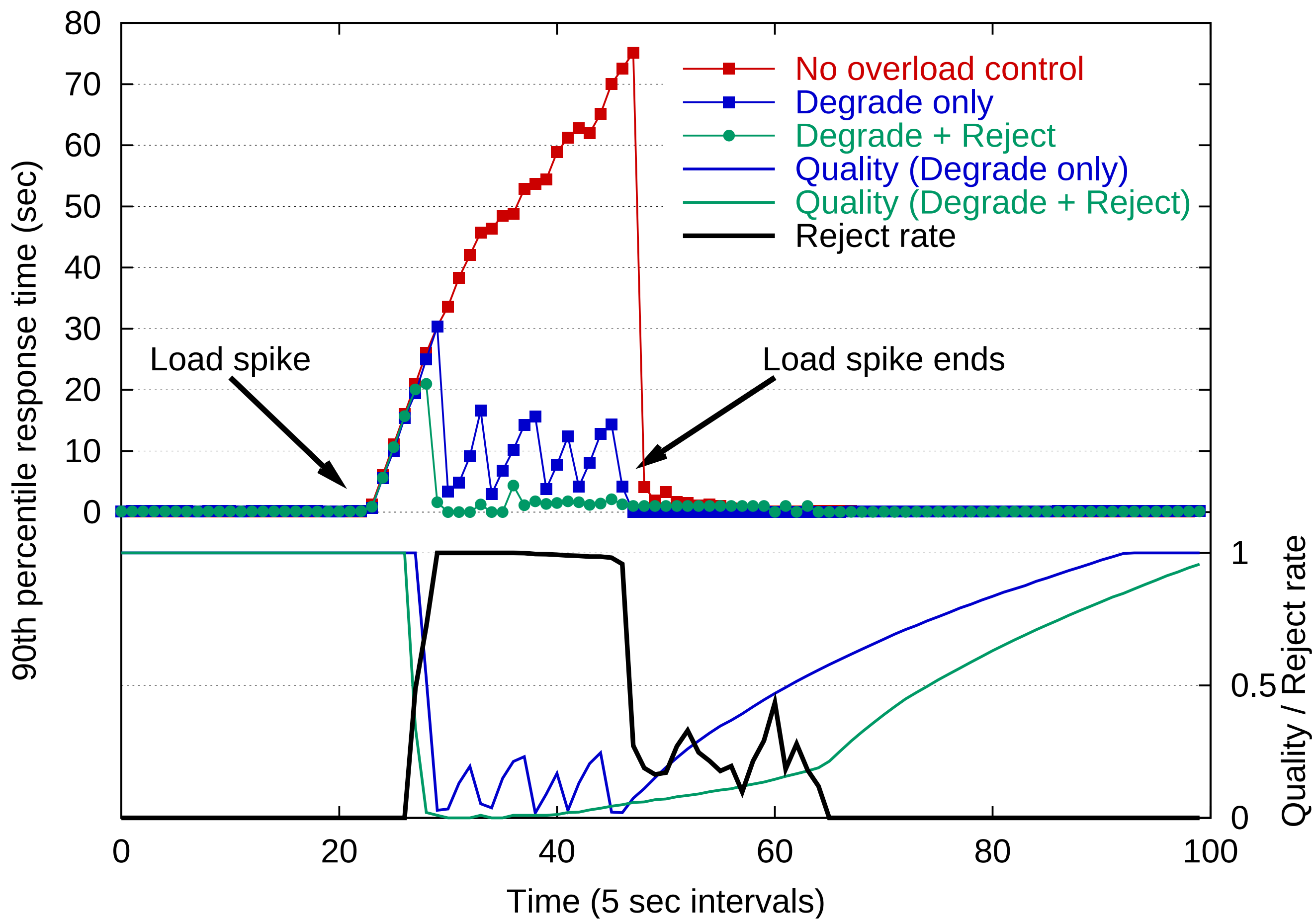
Response Time Controller Operation



Adjust incoming token bucket using AIMD control

- Target response time **1 second**
- Sample response times of requests through stage
- After 100 samples or 1 second:
  - ▷ *Sort measurements and measure 90th percentile*
  - ▷ *If 90th RT $< 0.9\times$ target RT, add $f(\text{err})$ to rate*
  - ▷ *If 90th RT $>$ target RT, divide rate by 1.2*

Overload management using service degradation



Degrade fidelity of service in response to overload

- Artifical benchmark: Stage crunches numbers with a varying “quality level”
- Stage performs AIMD control on service quality under overload
- Enable/disable admission controller based on response time and quality

Related Overload Management Techniques

Dynamic listen queue thresholding [Voigt, Cherkasova, Kant]

- Threshold or token-bucket rate limiting of incoming SYN queues
- Problem: Dropping or limiting TCP connections is bad for clients!

Specialized scheduling techniques [Crovella, Harchol-Balter]

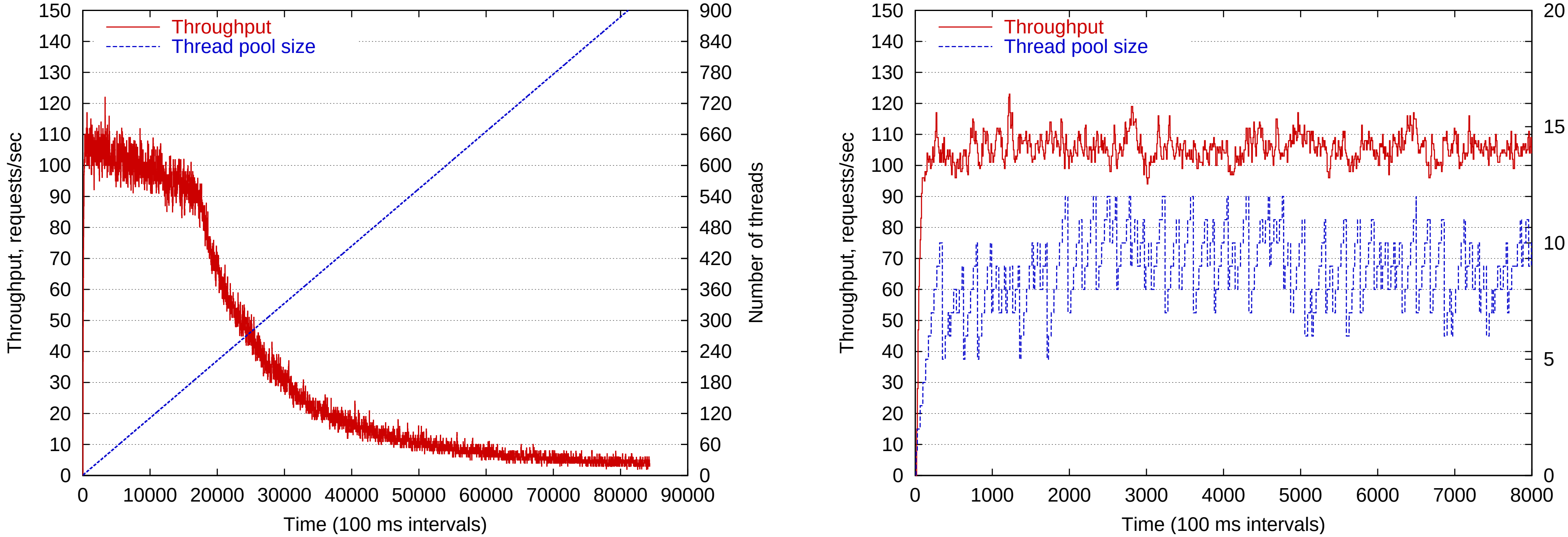
- e.g., Shortest-connection-first or Shortest-remaining-processing-time
- Often assumes 1-to-1 mapping of client request to server process

Class-based service differentiation [Bhoj, Voigt, Reumann]

- Kernel- and user-level techniques for classifying user requests
- Sometimes requires pushing application logic into kernel
- Adjust connection/request acceptance rate per class
    - ▷ *No feedback - static assignment acceptance rates*

We argue that overload management should be an **application design primitive** and not simply tacked onto existing systems

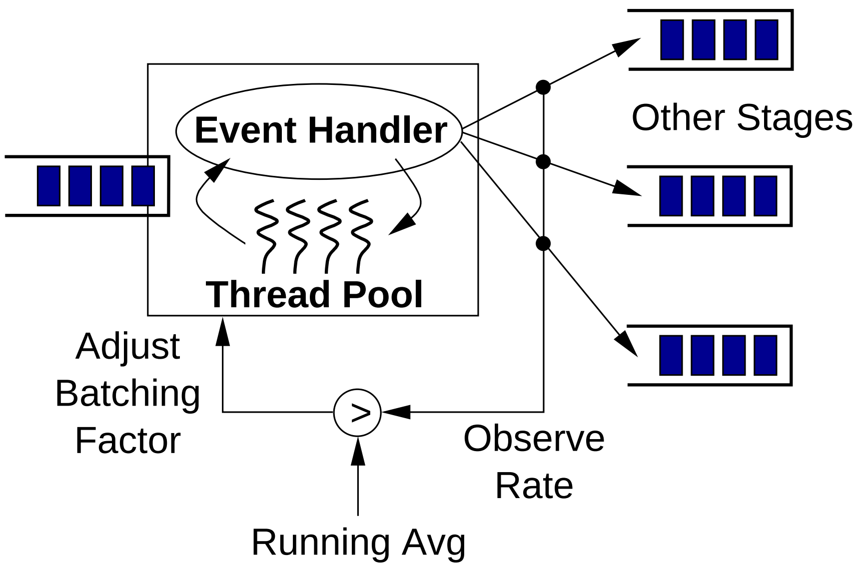
Thread pool thrashing detection

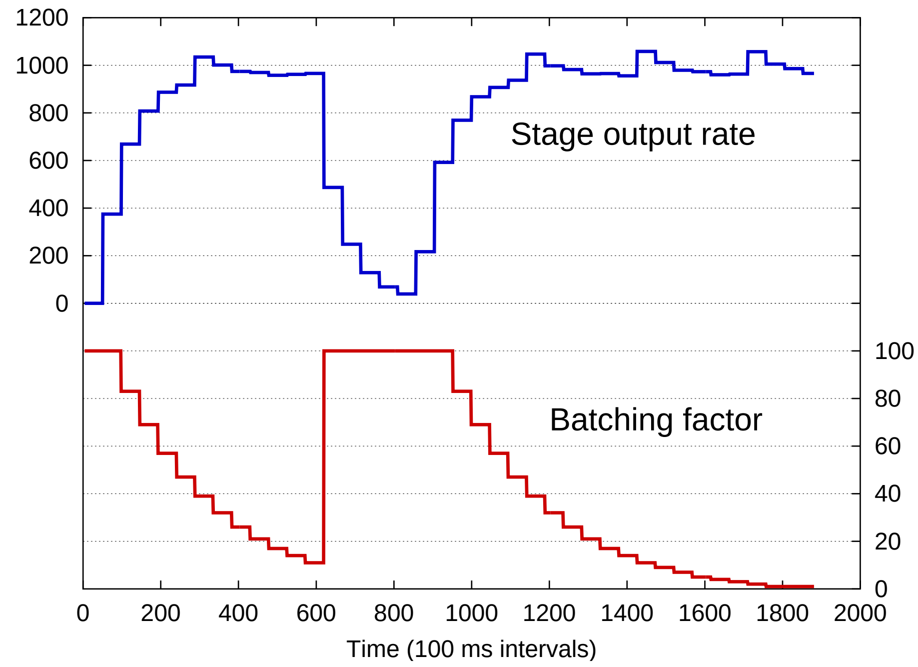


Detect maximum number of threads per stage

- Maintain moving average of thread pool size and throughput
- If new throughput $\geq$ saved throughput:
  - ▷ *Save current thread pool size*
- If new throughput $\leq 1.2\times$ saved throughput:
  - ▷ *Revert to saved pool size, minus "penalty" of 0-4 threads*

SEDA Batching Controller





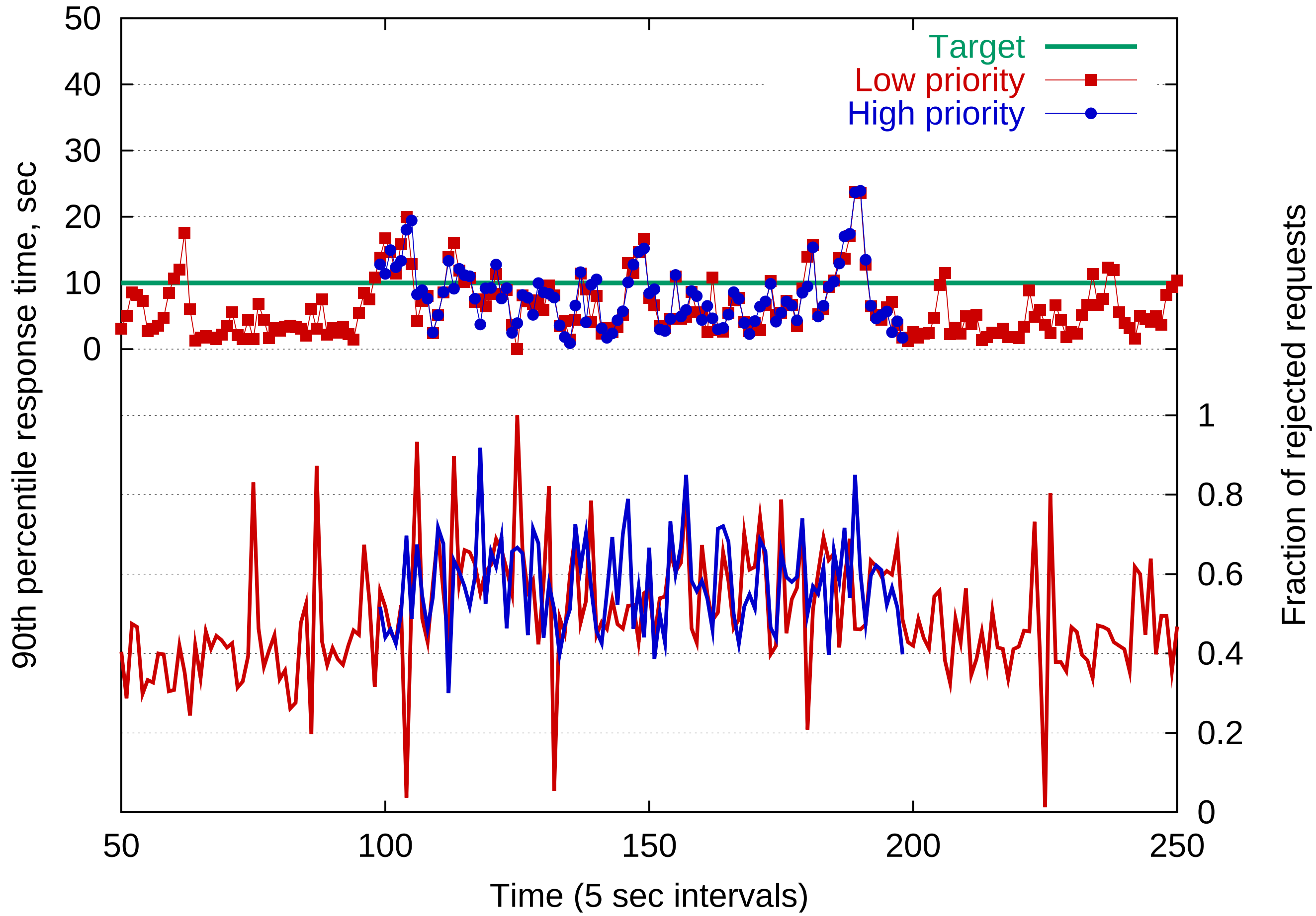
Goal: *Schedule for low response time and high throughput*

- **Batching factor:** number of events consumed by each thread
- Large batching factor → more locality, higher throughput
- Small batching factor → more parallelism, lower response time

Attempt to find smallest batching factor with stable throughput

- Reduces batching factor when throughput high, increases when low

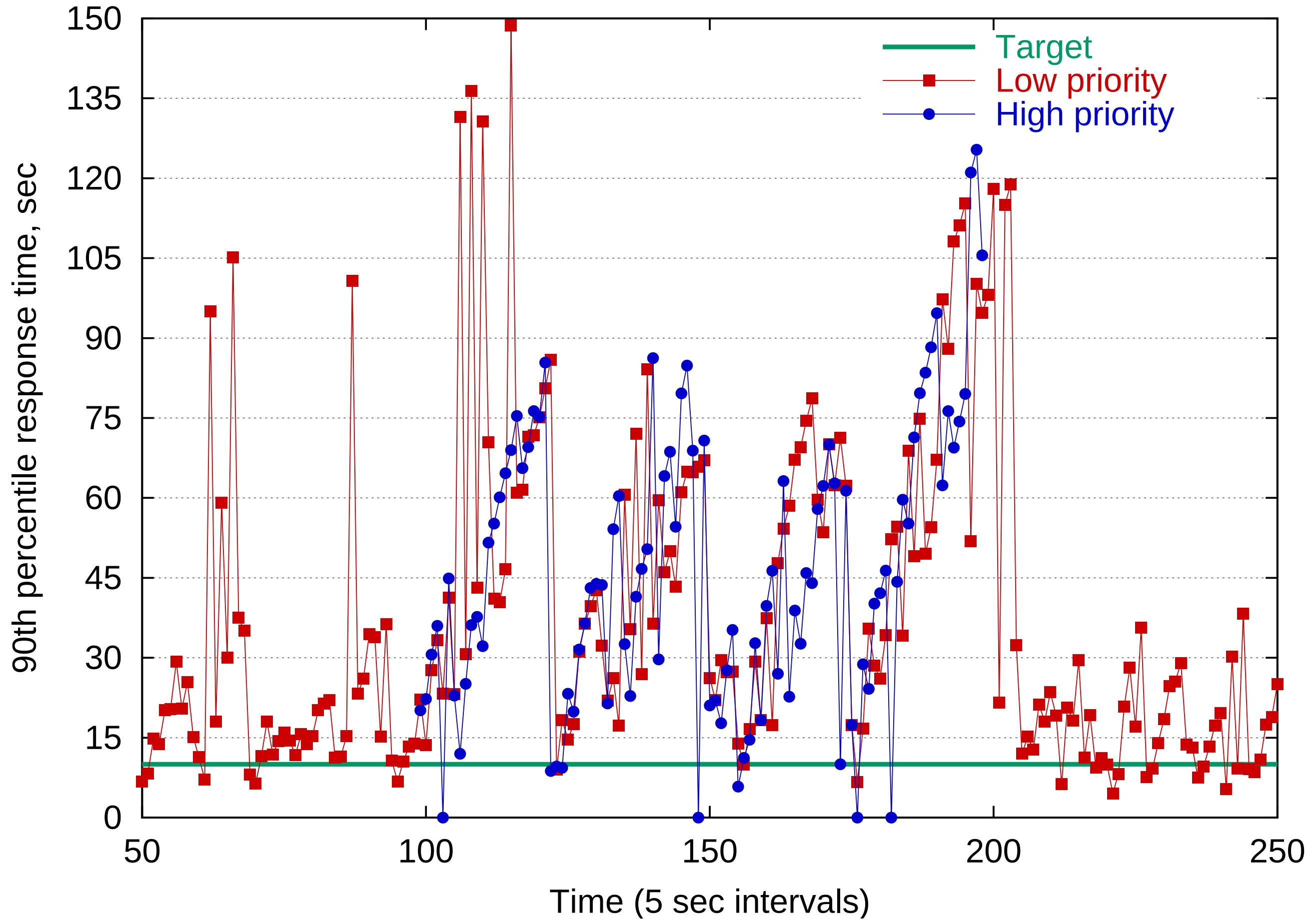
Without service differentation



Two classes of users with a 10 second response time target

- 128 users in each class
- No differentiation between classes of users
- High-priority users see same loss rate as low priority

Without response time control



Two classes of users without overload control enabled

- 128 users in each class
- Terrible response time performance

Lack of good evaluation environments

Very difficult to generate realistic open-loop loads!

- Don’t know how to characterize overload
- Need a large number of machines to crank up the load
  - ▷ *10x or 100x of typical experimental workloads*
- (All?) load generators degenerate to closed-loop case
  - ▷ *S-client [Banga et al.] only times out pending connections*

Need to study effects of overload in WAN environments

- ModelNet, Emulab, etc. are valuable emulation environments
  - ▷ *May need some validation under unusual conditions*
- PlanetLab: Deploying real wide-area testbed

Sandstorm's aSockets layer