Adaptive Overload Control for Busy Internet Servers

Matt Welsh

Intel Research and Harvard University

David Culler

Intel Research and U.C. Berkeley

The Problem: Overload in the Internet

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

Modern Internet services as highly dynamic

- Web servers do much more than serve up static pages
- e.g., server-side scripts (CGI, PHP), SSL, database access
- Requests have highly unpredictable CPU, memory, and I/O demands
- Makes overload very difficult to predict and manage

Some high-profile (and low-profile) examples of overload

- 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

Outline

- Traditional approaches to overload
- The Staged Event-Driven Architecture
- Adaptive overload control in SEDA
- Service differentiation and degradation
- Performance evaluation under massive load spikes
- Conclusions

Common approaches to 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

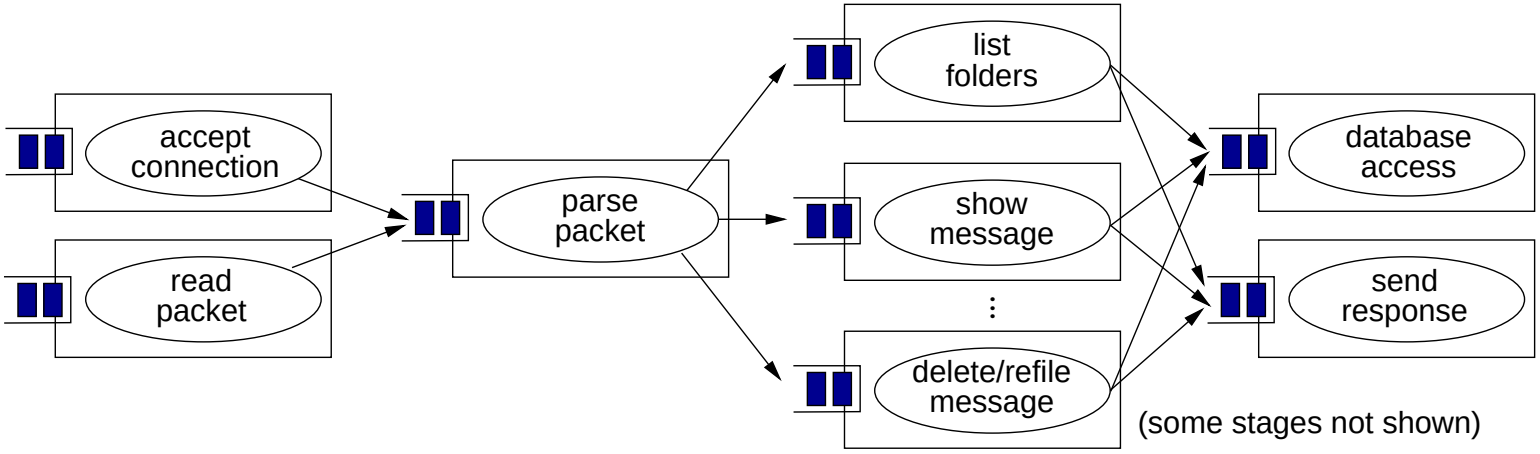
- e.g., Fixed limits on number of clients or CPU utilization
- Can underutilize resources (if limits set too low)
- or lead to oversaturation (if limits too high)

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.)

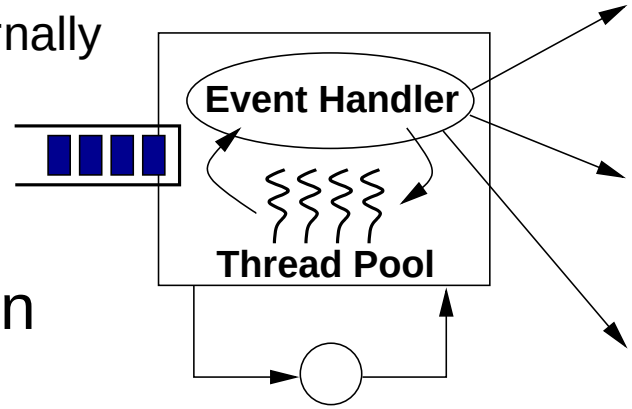
Many techniques studied only under simulation

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
- Each stage embodies a set of states from FSM
- Queues make load management explicit



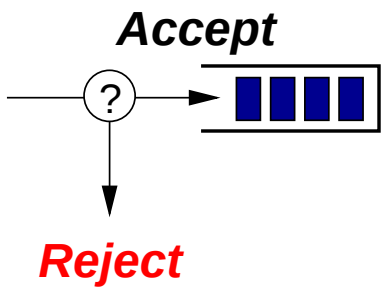
Stages contain a *thread pool* to drive execution

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

Apps don't allocate, schedule, or manage threads

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`)

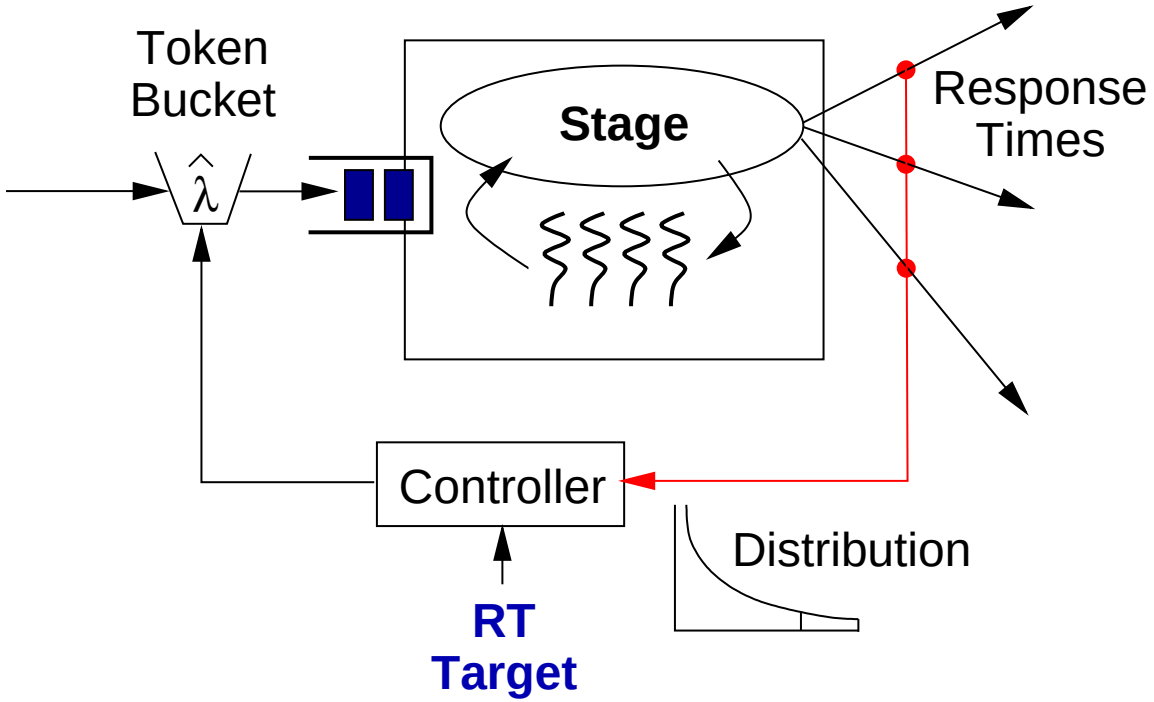
- Can couple with front-end load balancing across server farm

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

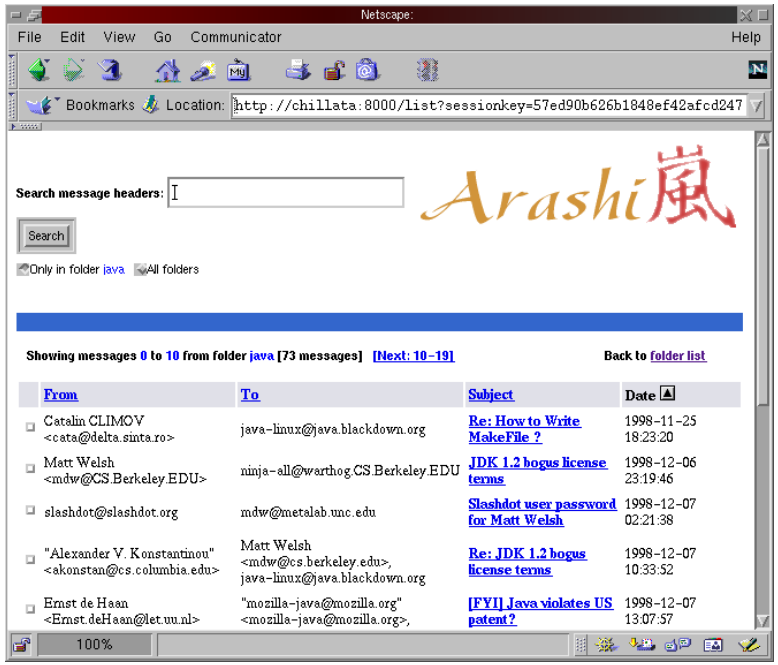
Arashi: A Web-based e-mail service

Yahoo Mail clone - “real world” service

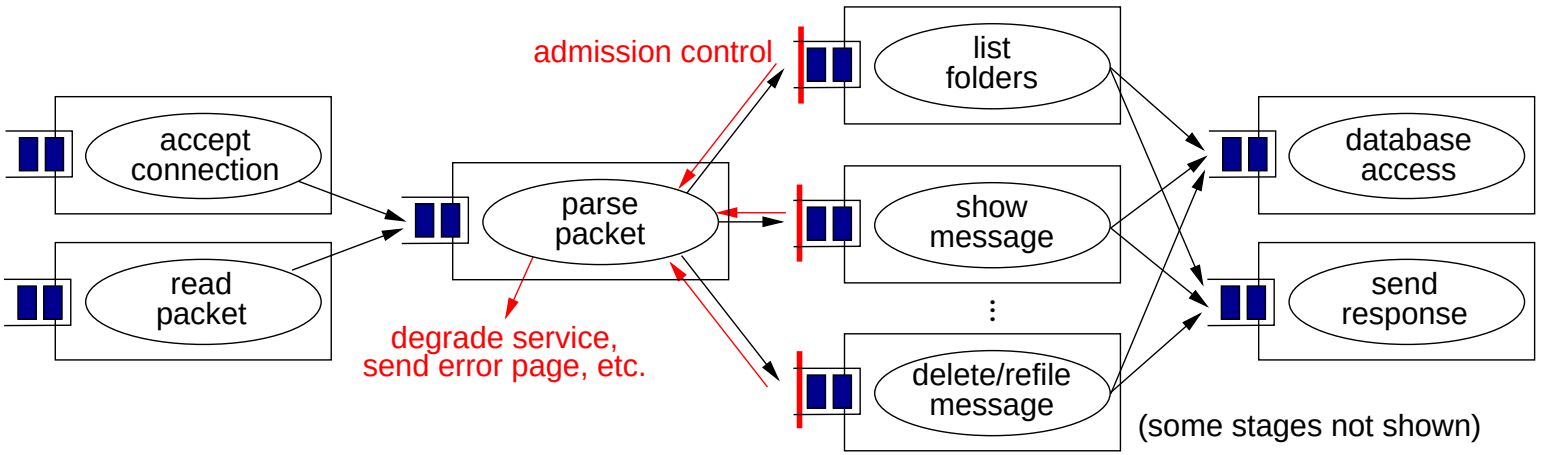
- Dynamic page generation, SSL
- New Python-based Web scripting language
- Mail stored in back-end MySQL database

Realistic client load generator

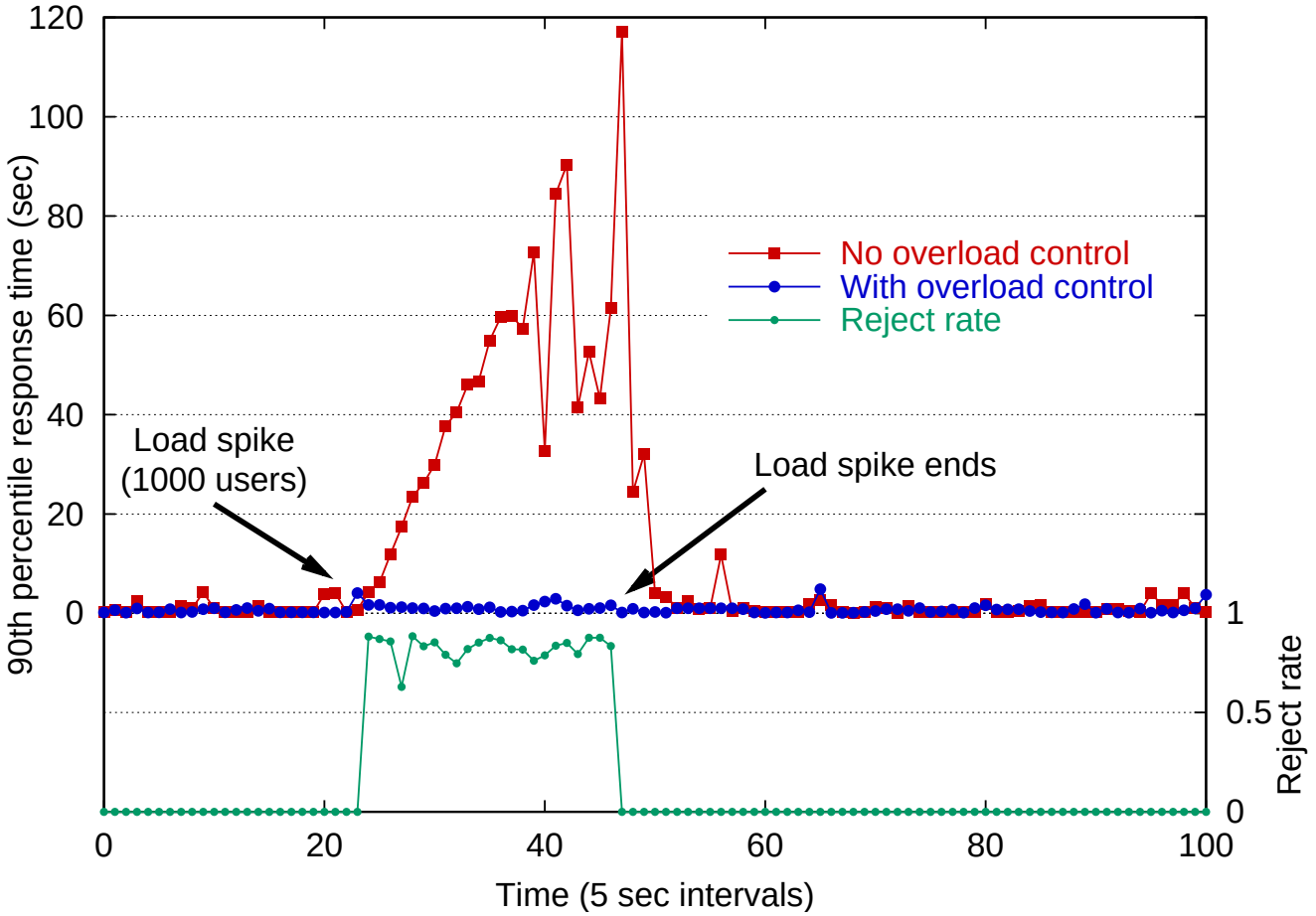
- Traces taken from departmental IMAP server
- Markov chain model of user behavior



Overload control applied to *each request type* separately:



Overload prevention during a load spike

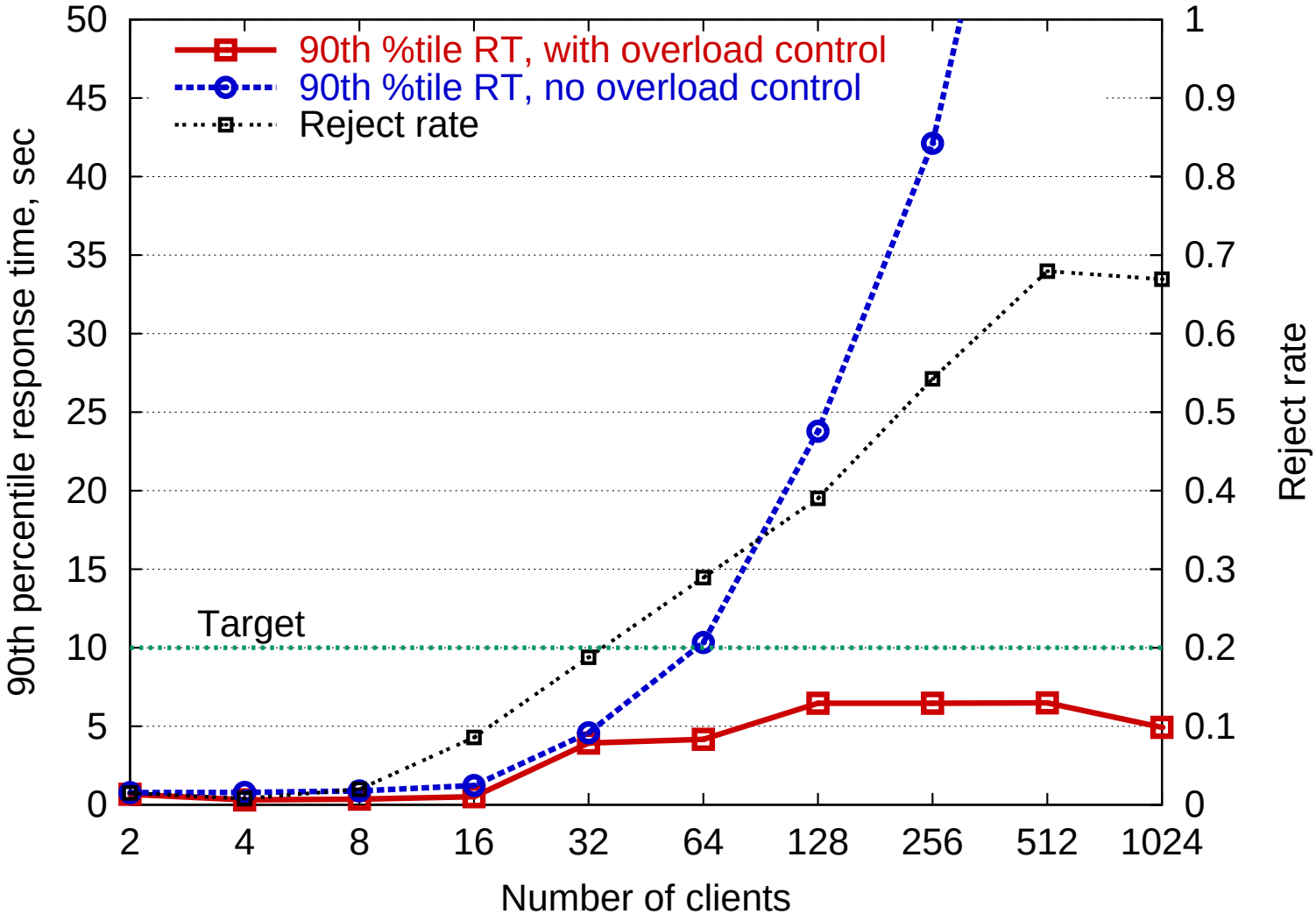


Sudden spike of 1000 users hitting Arashi service

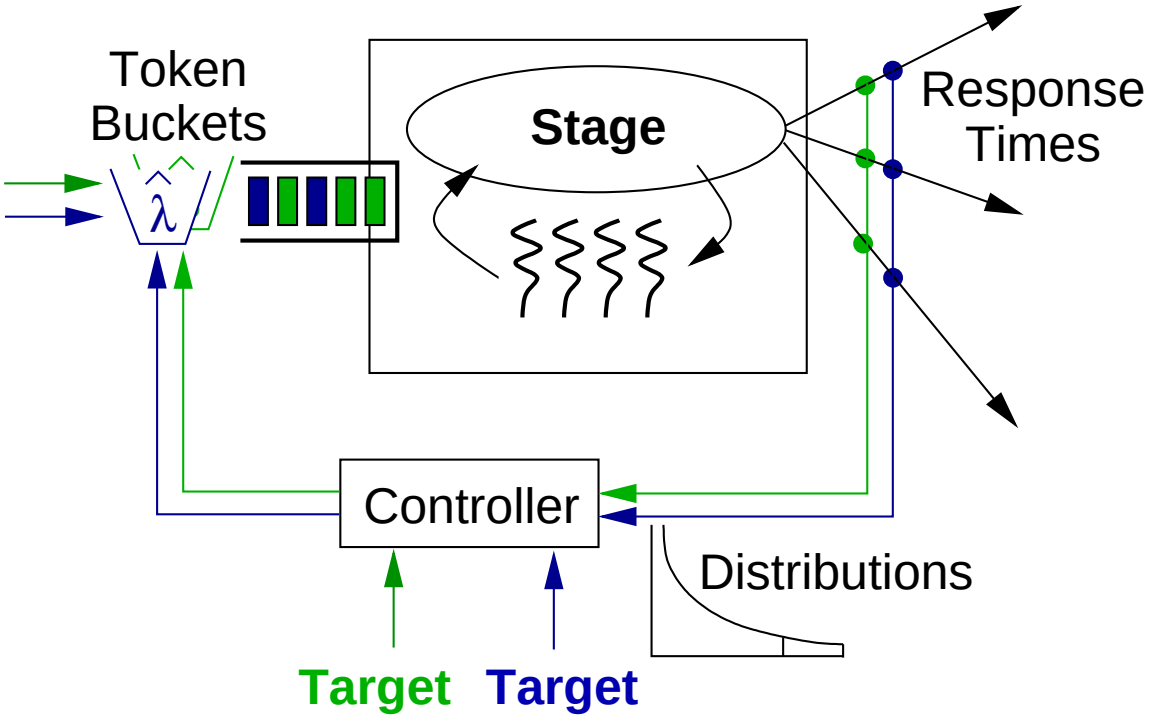
- 7 request types, handled by separate stages with overload controller
- 90th %tile response time target: **1 second**
- Rejected requests cause clients to pause for 5 sec

Overload controller has no knowledge of the service!

Overload control with scaling load



Service Differentiation



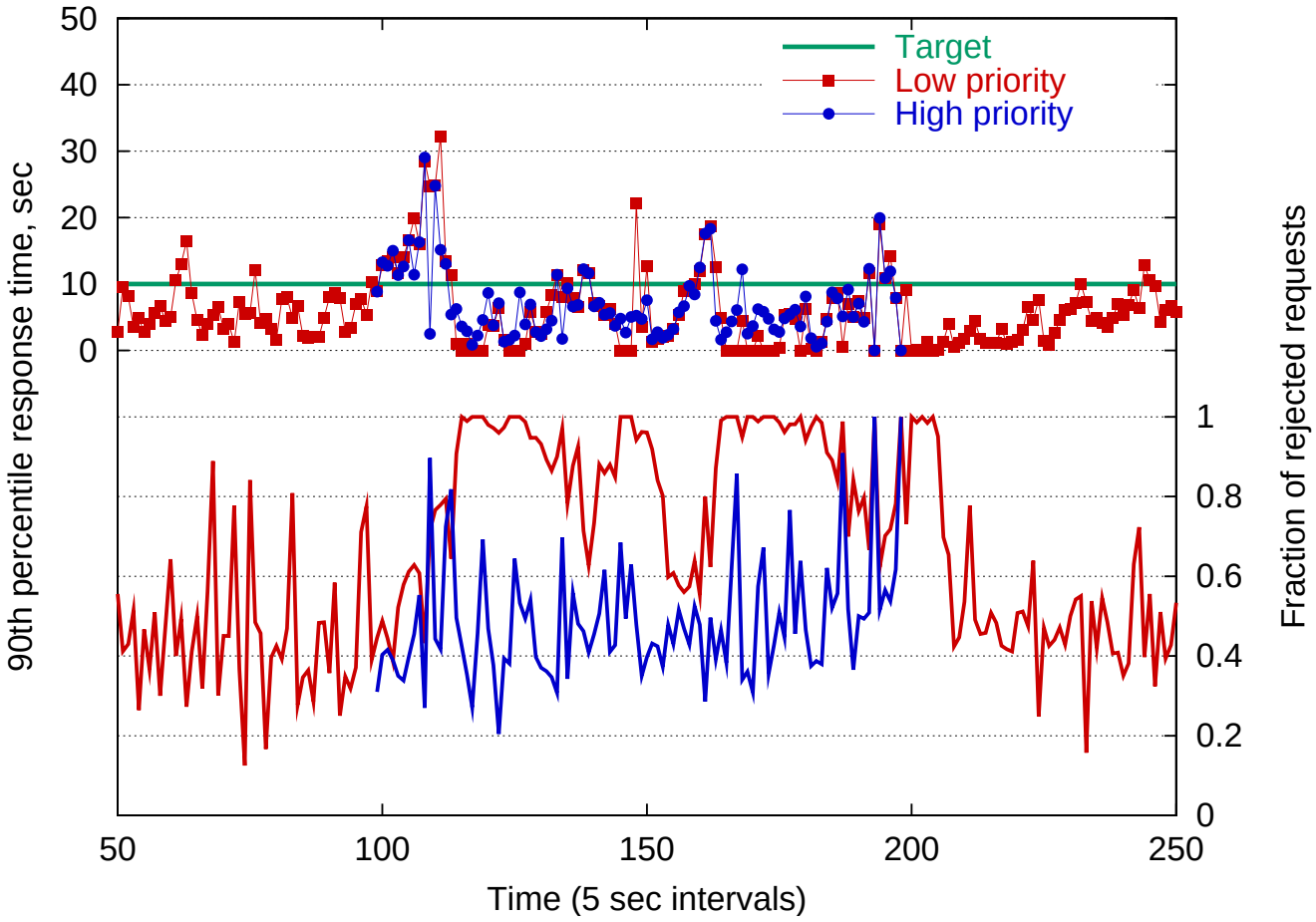
Differentiate users into multiple classes

- 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**

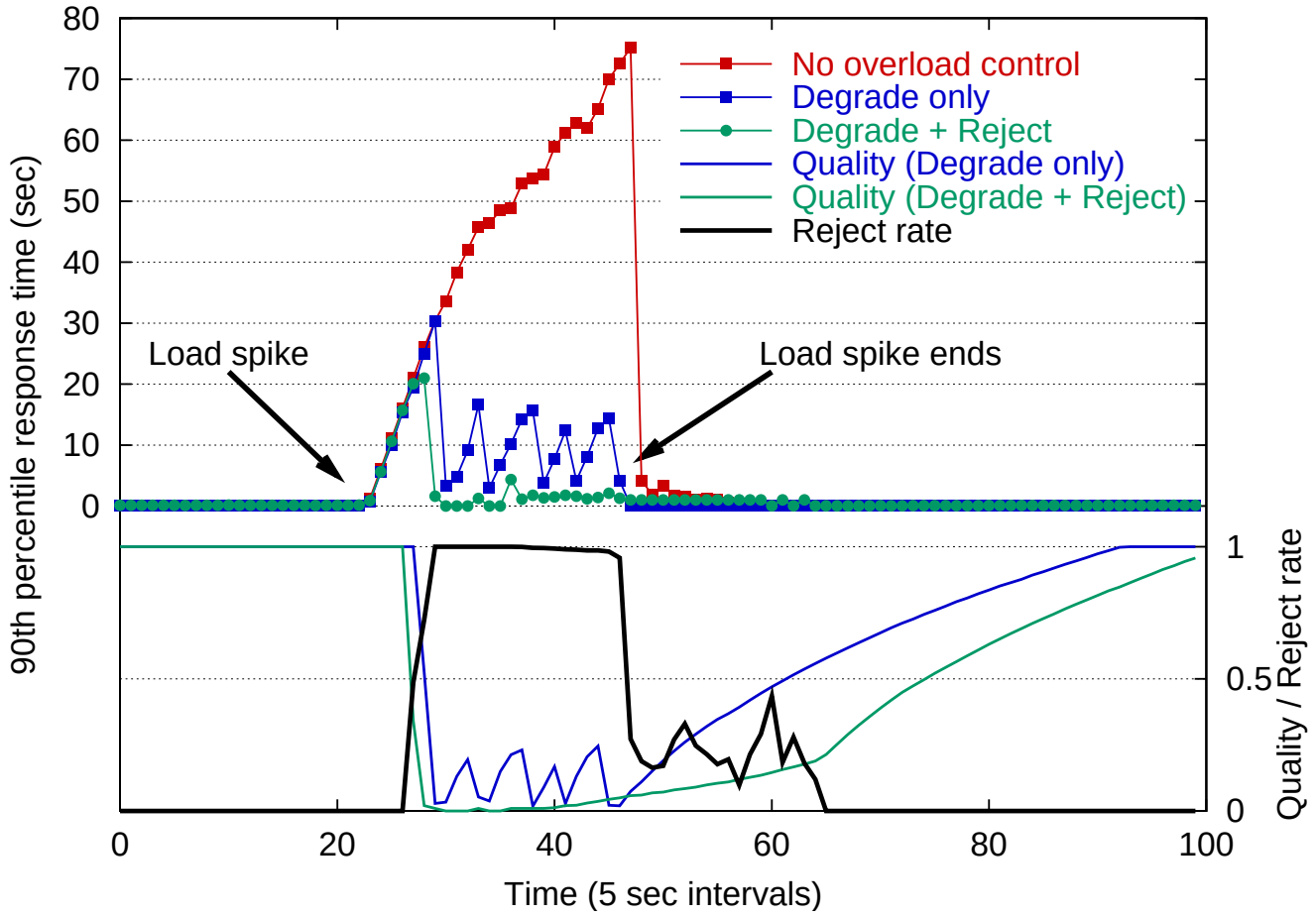
Service differentiation at work



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

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

Control theoretic resource management

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

Future Work

Automatic profiling, modeling, and tuning of overload controller

- Capture traces of stage performance vs. traffic and load mix
- Offline or online tuning of admission control parameters
- Use learning algorithms?

Extend local overload controller to global actions

- Adjust “front door” admission rate based on back-end bottleneck
- Prioritize stages that release resources
- Use global information, e.g., memory availability, to help

Further explore tradeoff between request rejection and degradation

- How to build general-purpose degradation into a service?
- Tie in with complex set of service level agreements

Summary

SEDA programming model exposes overload to the application

- Break event-driven application into stages connected with queues
- Queues can reject new requests - overload signal

Adaptive overload control at each stage

- Attempt to meet 90th-percentile response time target
- Adjust admission rate of each stage's queue
- Differentiated service using multiple admission controllers

Extensive evaluation in realistic overload setting

- Arashi service with highly dynamic behavior
- Realistic client load generator
- Evaluated overload control, service differentiation, and service degradation

For more information, software, and papers:

`http://www.cs.berkeley.edu/~mdw/proj/seda/`

Backup slides follow

Adaptive overload control algorithm

Monitor response time for each request in system

- Tag with current time on entry to service
- Gather distribution of accumulated response times at each stage

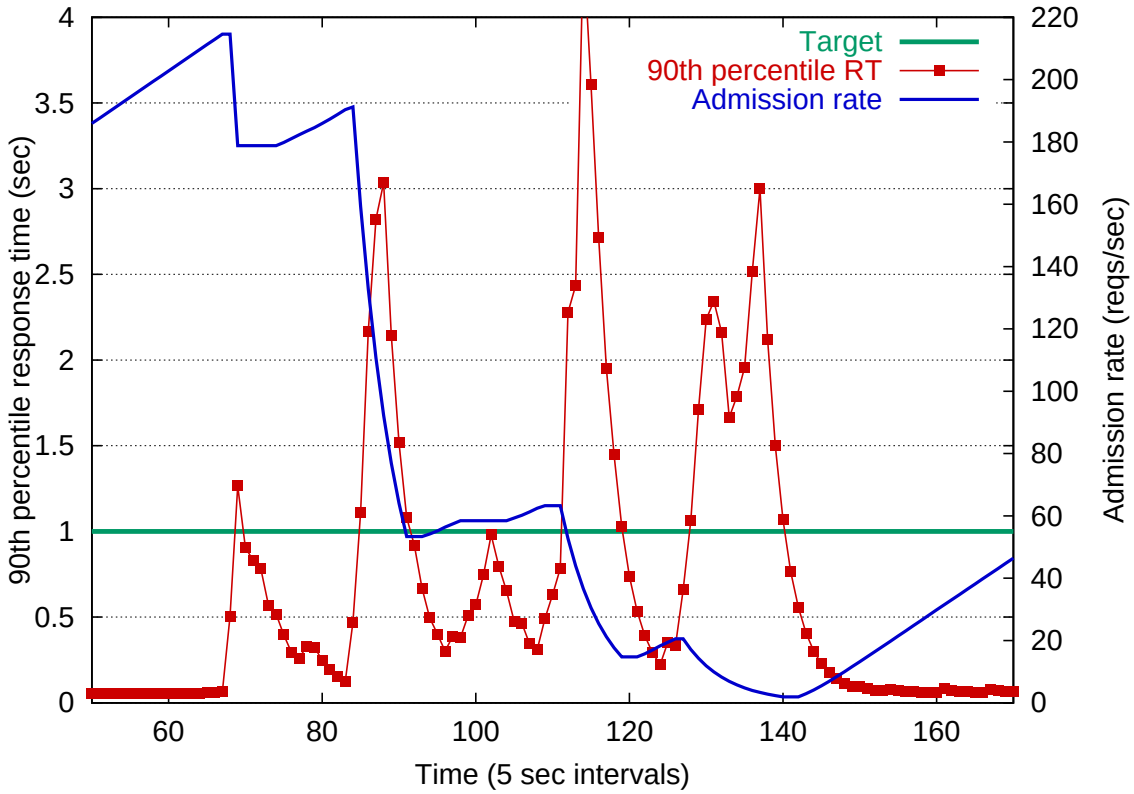
Controller adjusts admission rate of requests using token bucket

- Controller invoked after *N* requests processed or timeout
- EWMA filter used on 90th-percentile RT estimate
- Calculates error between current RT estimate and target
- If $err > err_d$, token bucket reduced by multiplicative factor: adj_d.
- If $err < err_i$, token bucket increased by additive factor: $-(err - c_i)adj_i$.

Parameters determined through extensive experimentation

- *N* = 100, timeout = 1 sec
- EWMA filter = 0.7
- err_i = -0.5, err_d = 0
- adj_i = 2.0, adj_d = 1.2

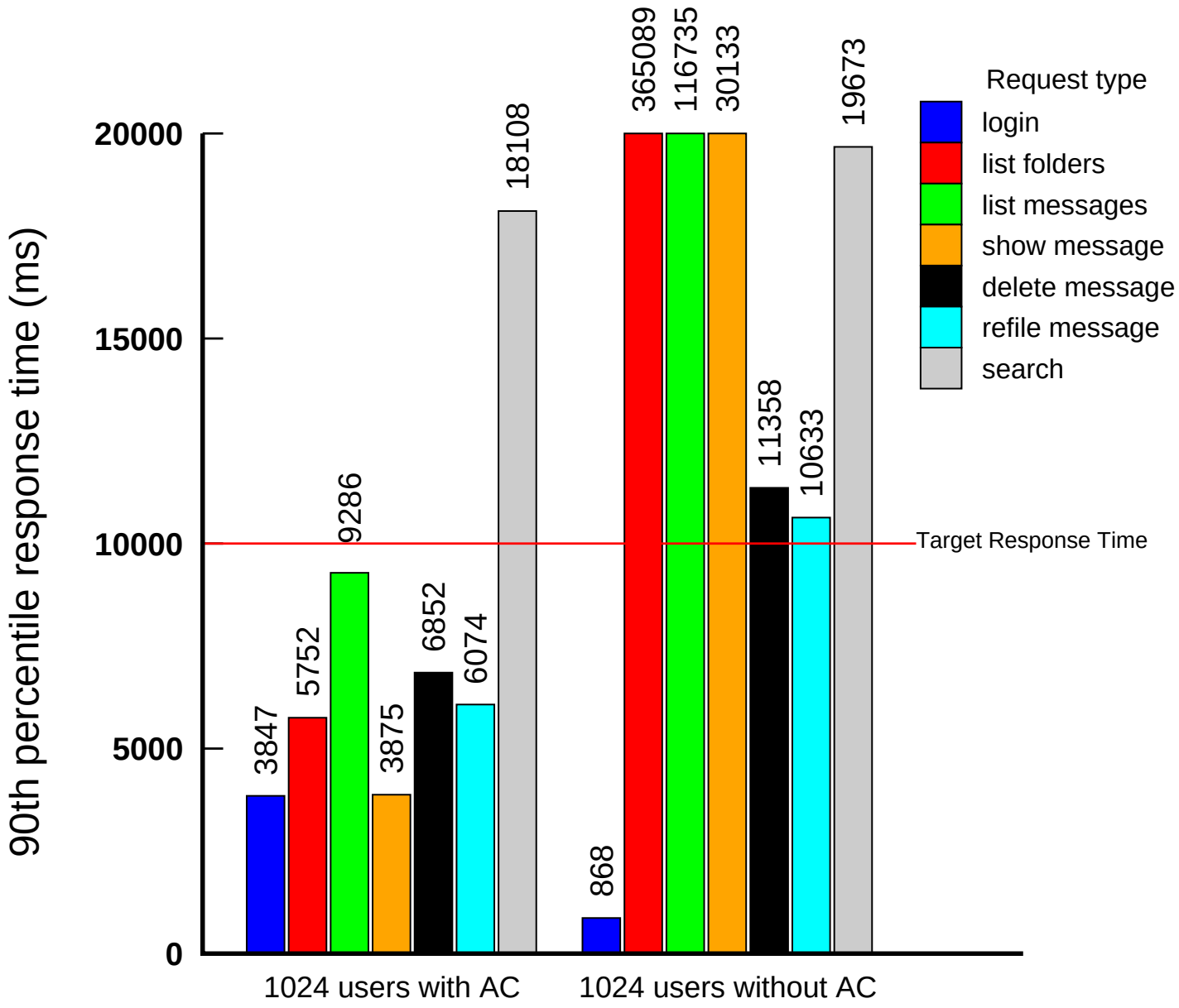
Response Time Controller Operation



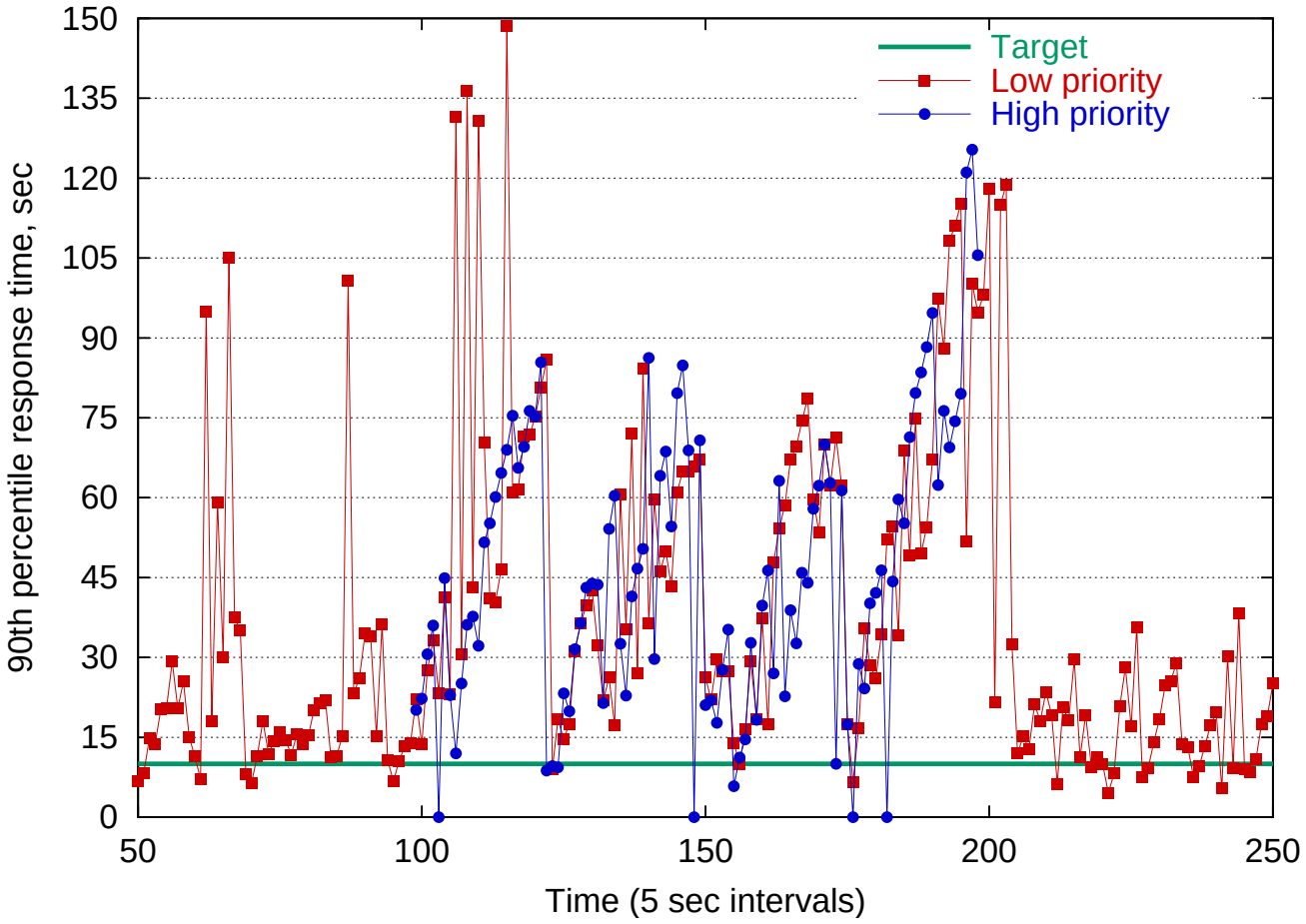
Adjust incoming token bucket using AIMD control

- 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* $f(err)$ *to rate*
  - *If 90th RT* $>$ *target RT, divide rate by* 1.2

Overload control by request type



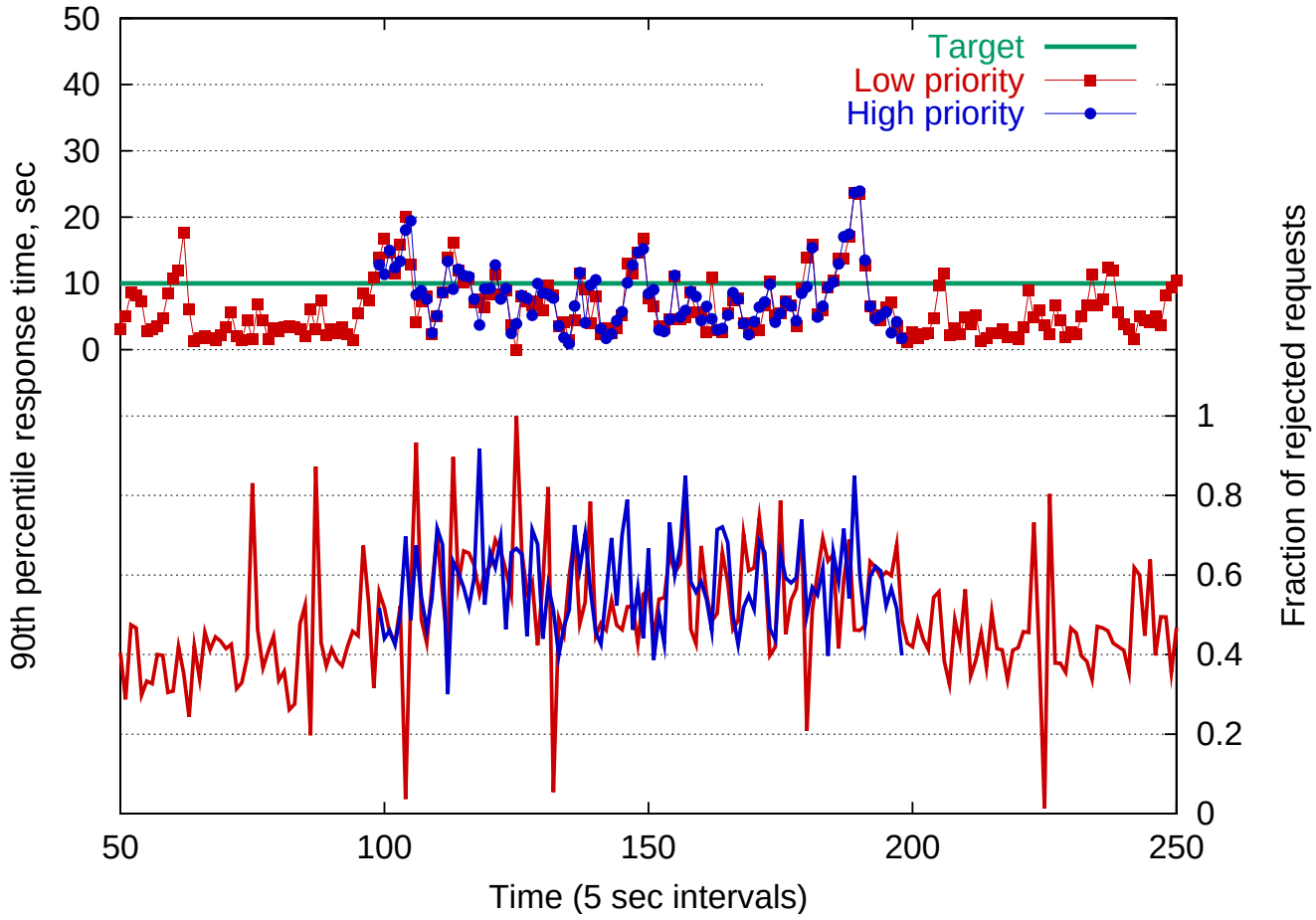
Without response time control



Two classes of users without overload control enabled

- 128 users in each class
- Terrible response time performance

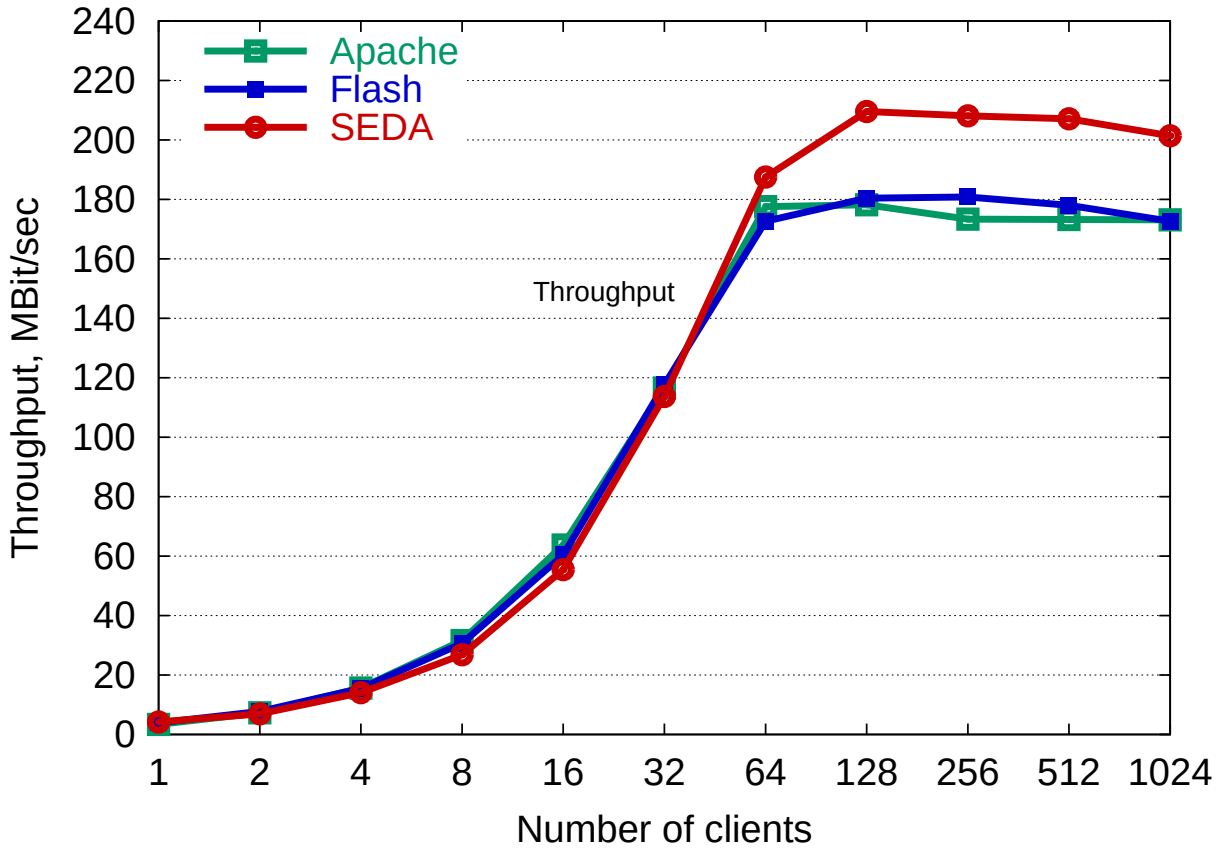
Without service differentation



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

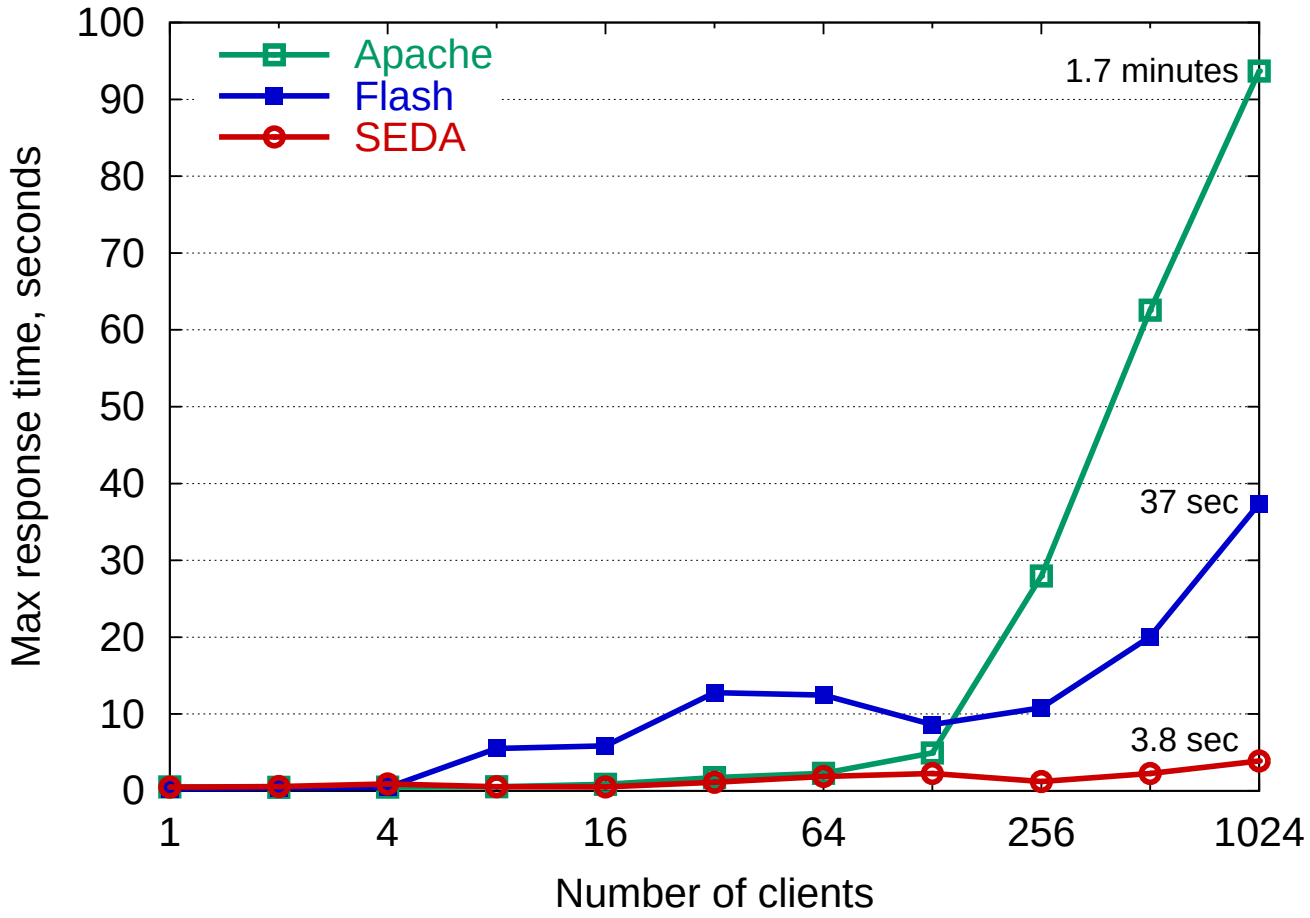
SEDA Scales Well with Increasing Load



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

User-level vs. kernel-level resource management

SEDA is a **user-level** solution: no kernel changes

- Runs on commodity systems (Linux, Solaris, BSD, Win2k, etc.)
- In contrast to extensive work on specialized OS, schedulers, etc.
- Explore resource control on top of imperfect OS interface
- “Grey box” approach - infer properties of underlying system from observed behavior

What would a SEDA-based “dream OS” look like?

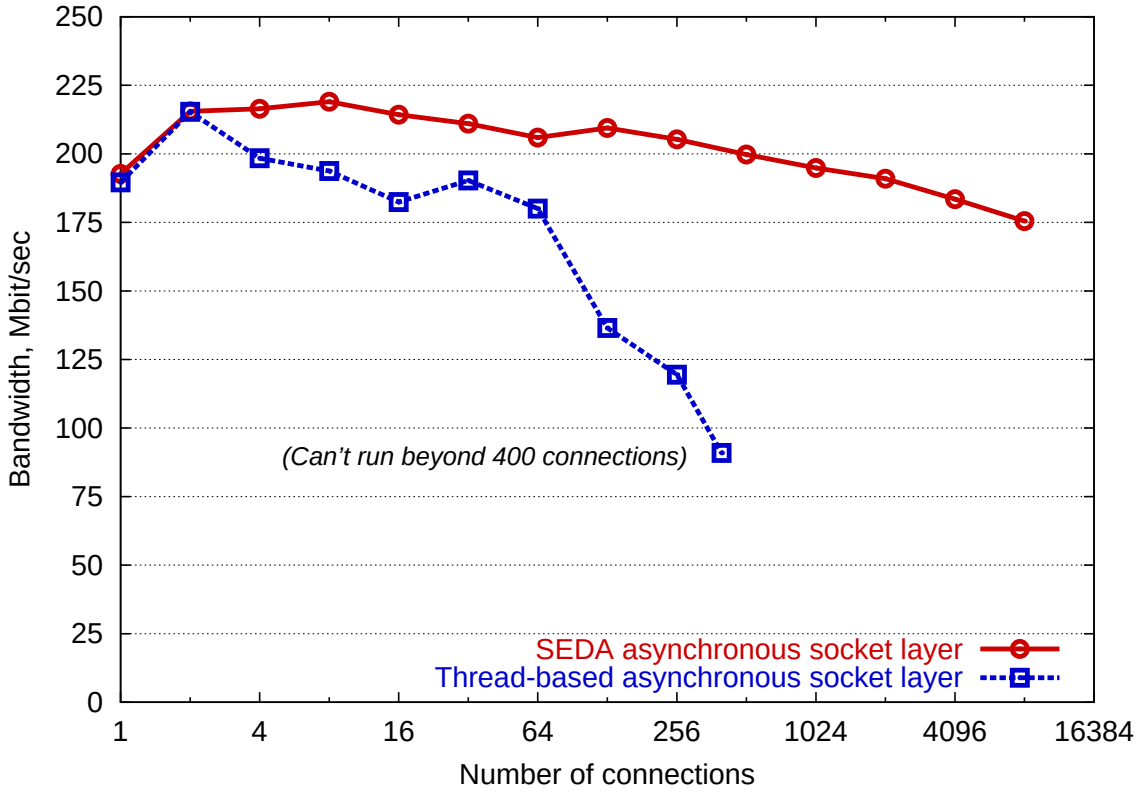
- Scalable I/O primitives: remove emphasis on blocking ops
- SEDA stage-aware scheduling algorithm?
- Greater exposure of performance monitors and knobs

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



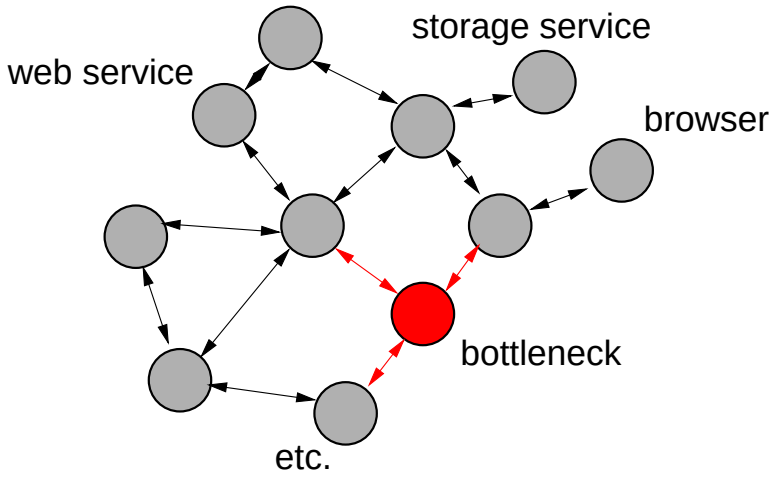
Distributed programming models and protocols

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



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!)