SEDA: Enabling Robust Performance for Busy Internet Servers

Matt Welsh

UC Berkeley Computer Science Division
mdw@cs.berkeley.edu

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

April 18, 2001

Internet Services Today

Massive concurrency demands

- Yahoo: 900 million+ pageviews/day
- Gartner Group: 127 million adult Internet users in US

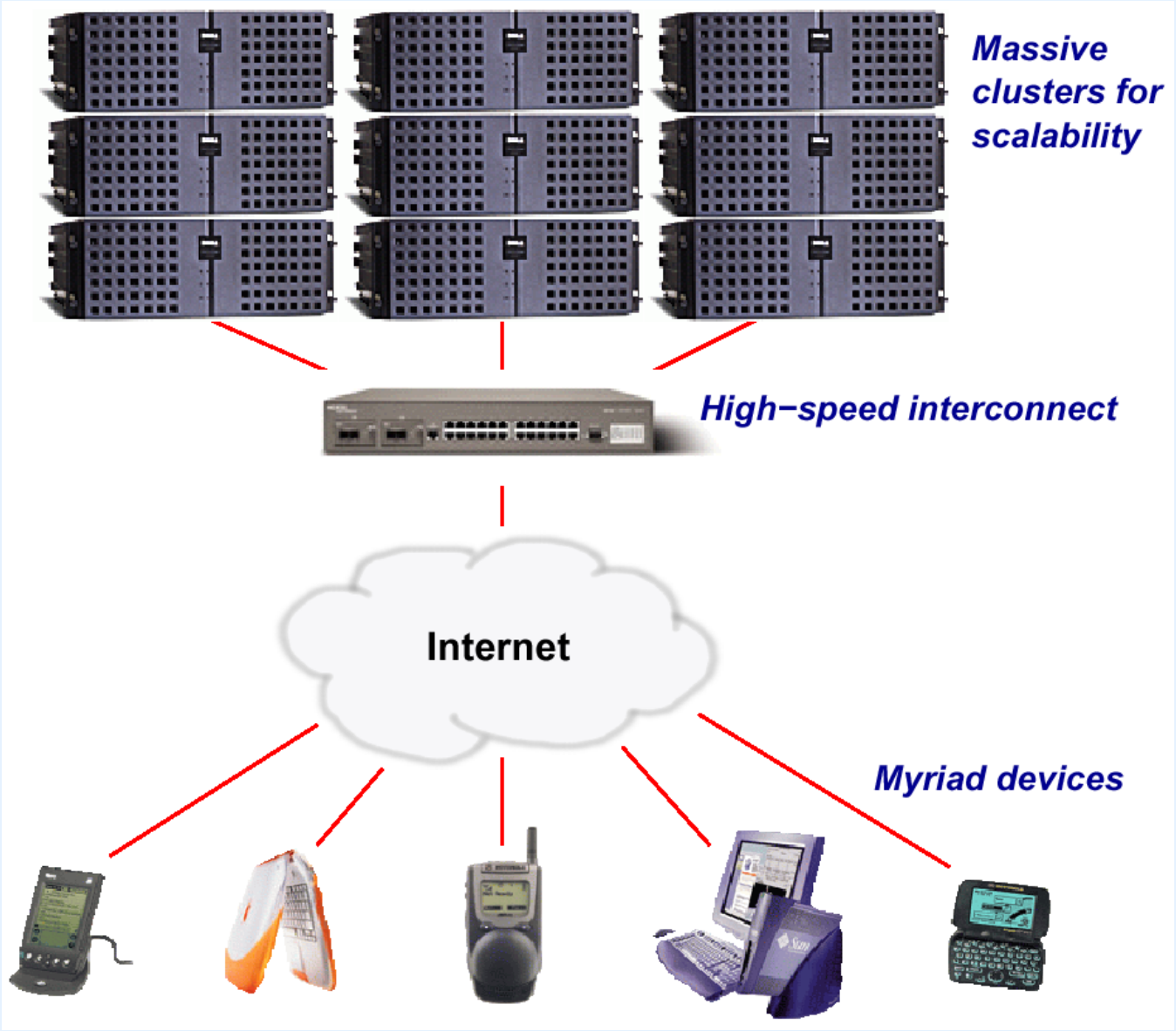
Must be extremely robust to load

- Peak load many times that of average
- Recent Presidential Election: 130% - 500% increase in news site traffic

Increasingly dynamic

- Days of the ''static Web'' are over
- Many sites based on dynamic content:
  - *e-Commerce, stock trading, driving directions, etc.*
- Application domains are expanding
  - *business-to-business, peer-to-peer*

Context: The Ninja Project Vision



New Directions in Service Design

Open infrastructure

- Allow users to easily push new services into the network
- Managed resources, protection, and scalability
- Reduce complexity of service authorship
  - *e.g., Build your own eBay in 100 lines of code*

Delivery to arbitrary devices

- Wide range of wired and wireless end devices on the horizon
- Network-embedded proxy customizes service interface to specific device
  - *e.g., Transcode email to/from voice for cell phone*

Service composition

- Potential to aggregate and compose services in the network
  - *e.g., Fetch email message and extract address; generate directions from present location*
- Strongly-typed interfaces and proxies are key

Importance of Load Conditioning

Availability is crucial

- No more than a few minutes of downtime a year
- Lawsuits are possible (e.g. E*Trade outage)

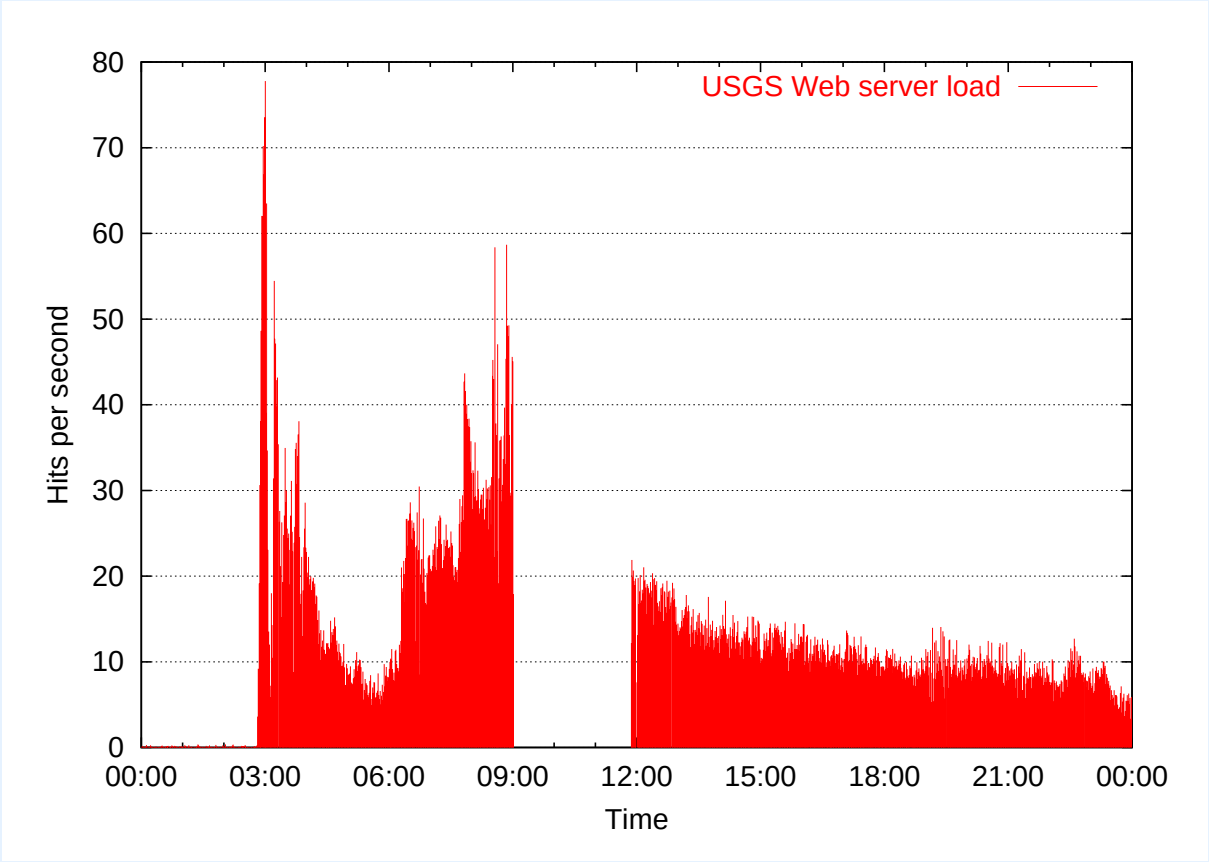
Overprovisioning is generally infeasible

- Peak demand many times that of average
- Service load is extremely bursty
  - ▷ *(cost of n machines) $>$ (n $\times$ cost of 1 machine)*
- Although replicated, services still experience huge load spikes

Must be well-conditioned to load

- Load spikes occur exactly when the service is most valuable!
- Service should not overcommit resources
- Performance should not degrade such that all clients suffer

The Slashdot Effect



Web log from USGS Pasadena Field Office

- M7.1 earthquake at 3 am on Oct 16, 1999
- Load increased 3 orders of magnitude in 10 minutes
- Disk log filled up at 9am

Problem Statement

Supporting massive concurrency is hard

- Traditional OS designs provide thread/process-based concurrency
- Designed for timesharing
  - ▷ *Entail high overheads and memory footprint*
- These mechanisms don't scale to many thousands of tasks

Existing OS designs do not provide graceful management of load

- Standard OSs strive for maximum resource transparency
- But, Internet services require extensive control

Few tools aid the development of scalable services

- Much work on performance and robustness engineering for specific services
  - ▷ *e.g., Fast, event-driven Web servers*
- As services become more dynamic, this engineering burden is excessive
- Need general-purpose toolkit for service construction

Proposal: The Staged Event-Driven Architecture

SEDA: A new architecture for Internet services

- Combines aspects of threads and event-driven programming
- Break applications into *stages* separated by *queues*

Simplify task of building highly-concurrent applications

- SEDA design supports massive concurrency
- Use of stages supports modularity, code reuse, debugging

Enable load conditioning

- Event queues allow inspection of request streams
- Can perform prioritization or filtering during heavy load

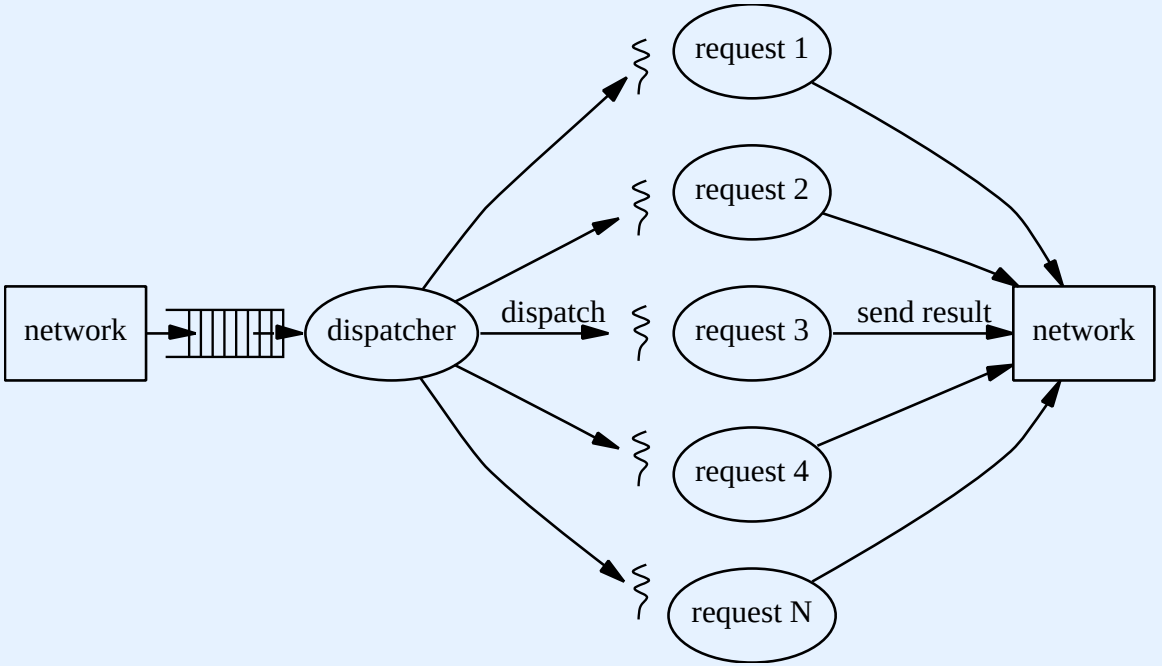
Self-tuning resource management

- Shield apps from complexity of thread management, event scheduling, and I/O
- Built-in scalable network and disk I/O layers
- *Dynamic resource controllers* adapt runtime parameters

Outline of this Talk

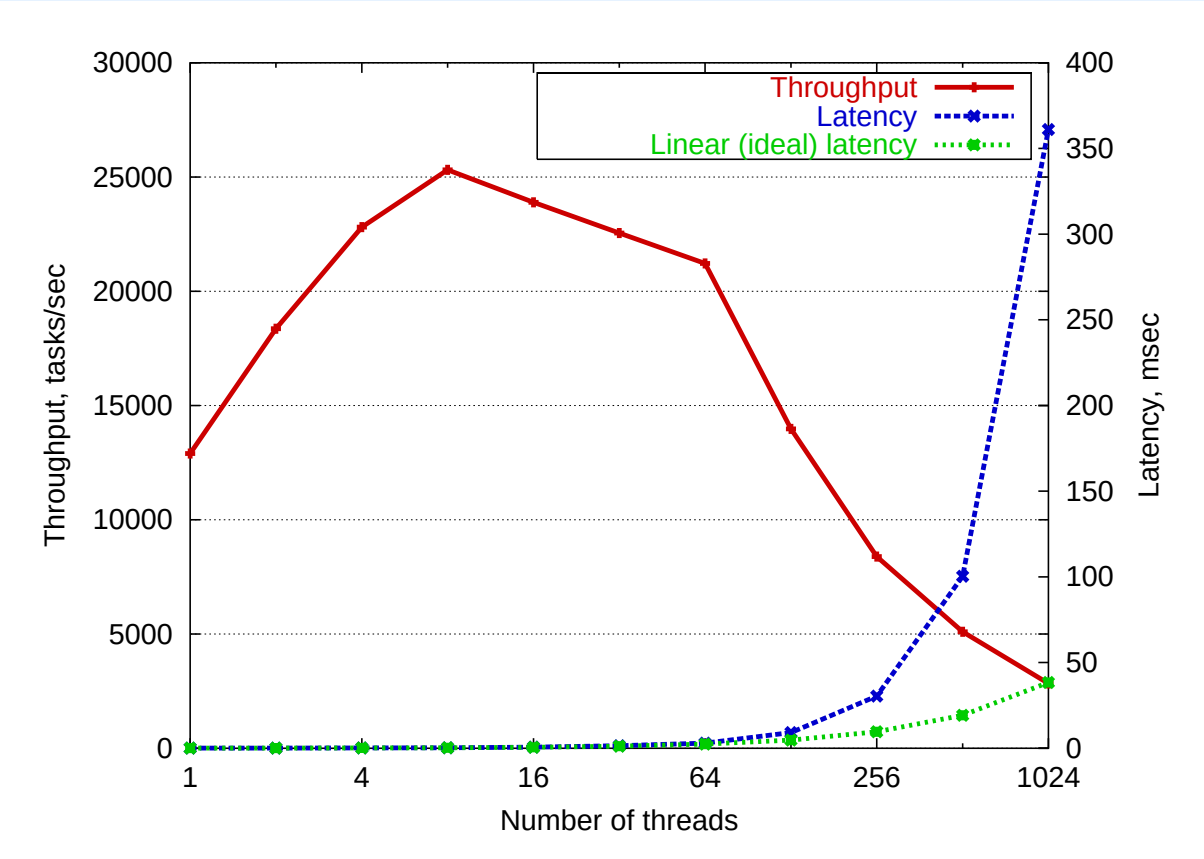
- Problems with Existing Concurrency Models
- The SEDA Architecture
- Dynamic Resource Controllers
- SEDA Asynchronous I/O Primitives
- Application Evaluation: HTTP and Gnutella Servers
- Future Work and Conclusions

Thread-Based Concurrency



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

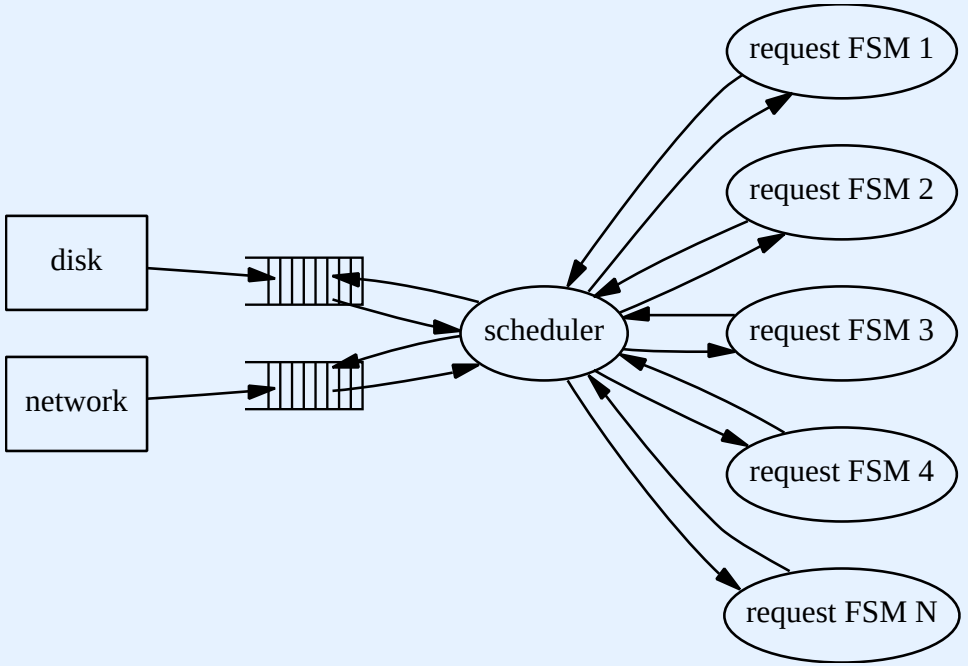
Problems with Threads



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

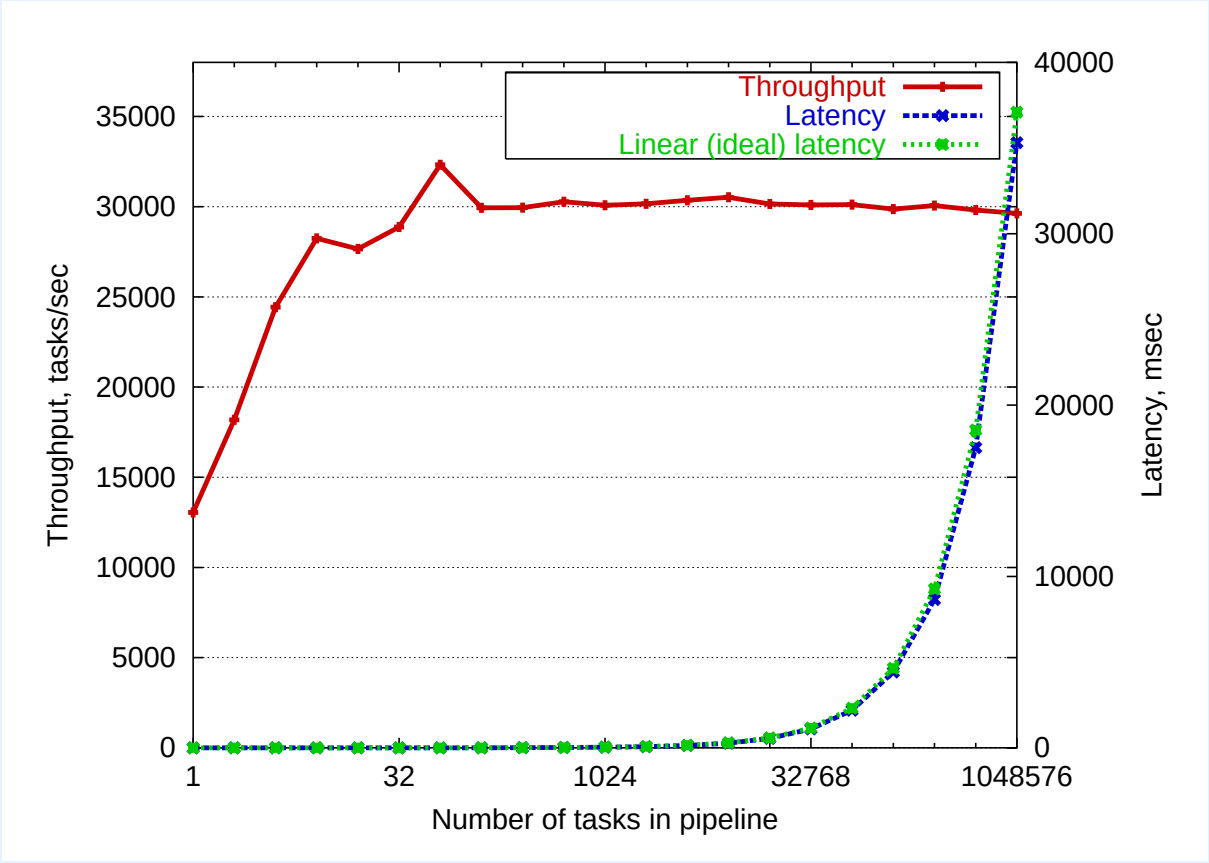
- High resource usage (stacks, etc.)
- High context switch overhead, contended locks expensive
- Too many threads $\rightarrow$ throughput meltdown, response time explosion

Event-based Concurrency



- Single thread processes events
- Each concurrent flow implemented as a finite state machine
- Application controls concurrency directly
  - *Must schedule events and FSMs carefully*
  - *Often very application-specific*

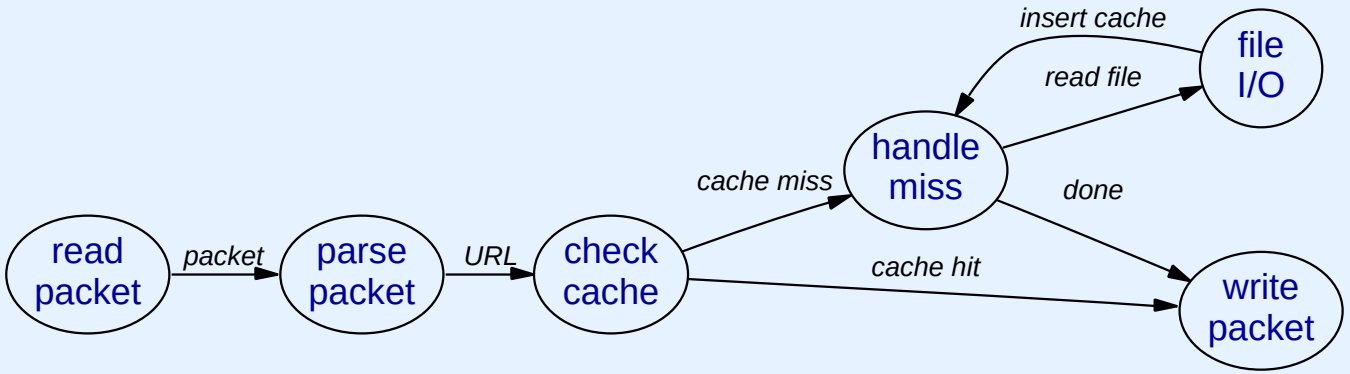
Well-Conditioned Performance



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

- Throughput saturates as load increases
- Response time increases linearly

''Monolithic'' Event-driven Server



One FSM per HTTP request

- Single thread processes all concurrent requests disjointly

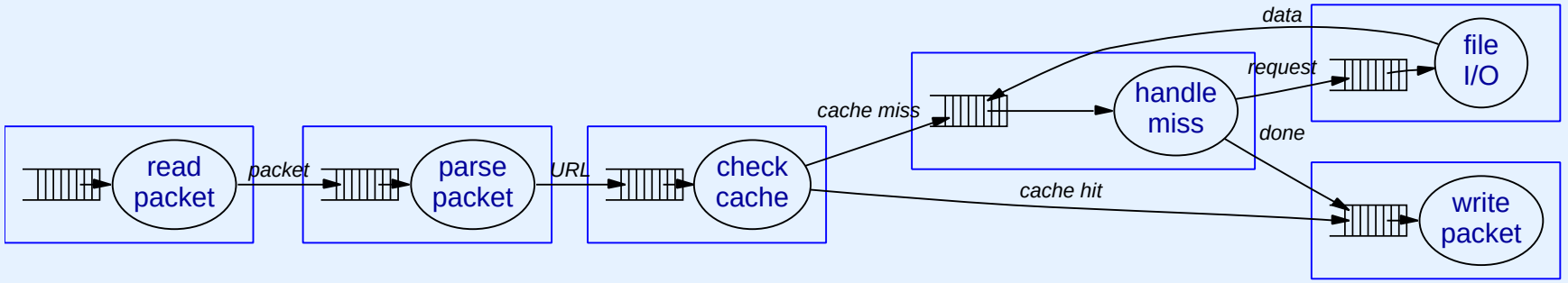
FSM code can never block

- But page faults, garbage collection force a block

Difficult to modularize

- Code for each state highly interdependent

Staged Event-Driven Architecture (SEDA)



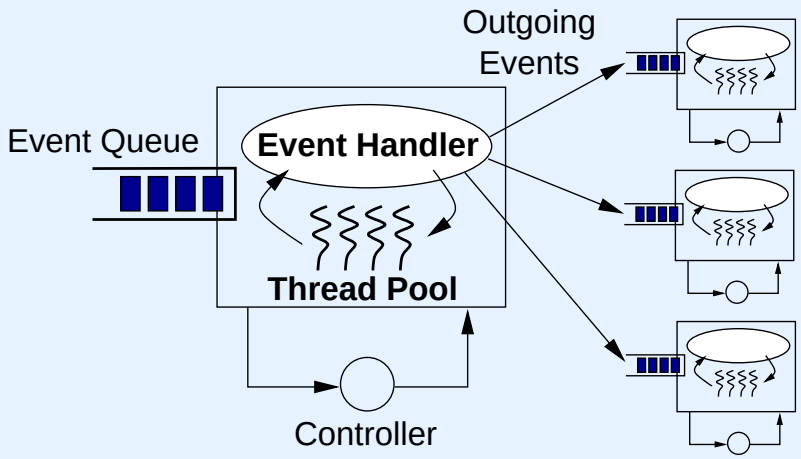
Decompose service into *stages* separated by *queues*

- Each stage embodies a set of states from FSM
- Queues introduce control boundary for isolation

Threads used to drive stage execution

- Decouples event handling from thread allocation and scheduling
- Stages may run in sequence or in parallel
- Stages may block internally
  - ▷ *Devote small number of threads to a blocking stage*

Stages as Robust Building Blocks



Application logic embodied as **event handler**

- Receives multiple events, processes them, enqueues outgoing events
- No direct control over event queues or threads
- Event queue absorbs excess load, bounded thread pool maintains concurrency

Stage controller

- Manages resource allocation and scheduling
- Controls number and ordering of events passed to handler
- Event handler may internally drop, filter, reorder events

Applications as Network of Stages

Event queues are finite

- Enqueue operation may fail if queue rejects new entries
- Backpressure implemented by blocking on full queue
- Load shedding implemented by dropping events
  - ▷ *May also take alternate action, e.g., degraded service*

Event queues decouple stage execution

- Introduces explicit control boundary
  - ▷ *Threads may only execute within a single stage*
- Provides isolation, modularity, independent load management
- Core composition question: function call or event queue?

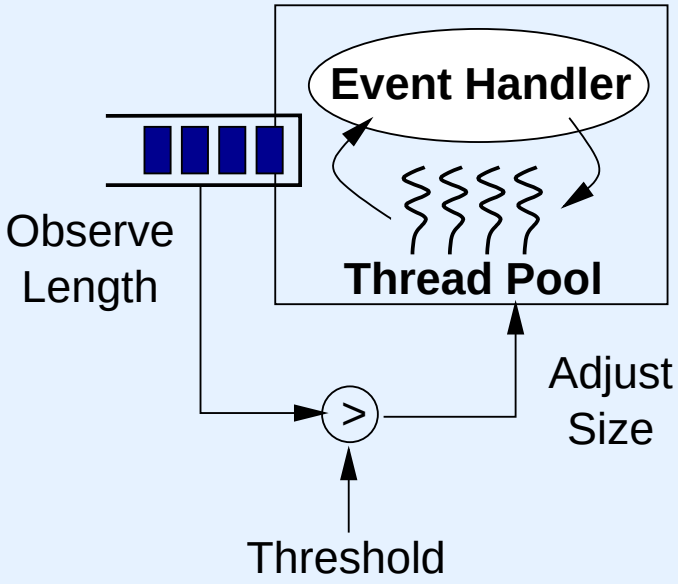
Facilitates debugging and profiling

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

Outline of this Talk

- Problems with Existing Concurrency Models
- The SEDA Architecture
- Dynamic Resource Controllers
- SEDA Asynchronous I/O Primitives
- Application Evaluation: HTTP and Gnutella Servers
- Future Work and Conclusions

SEDA Thread Pool Controller



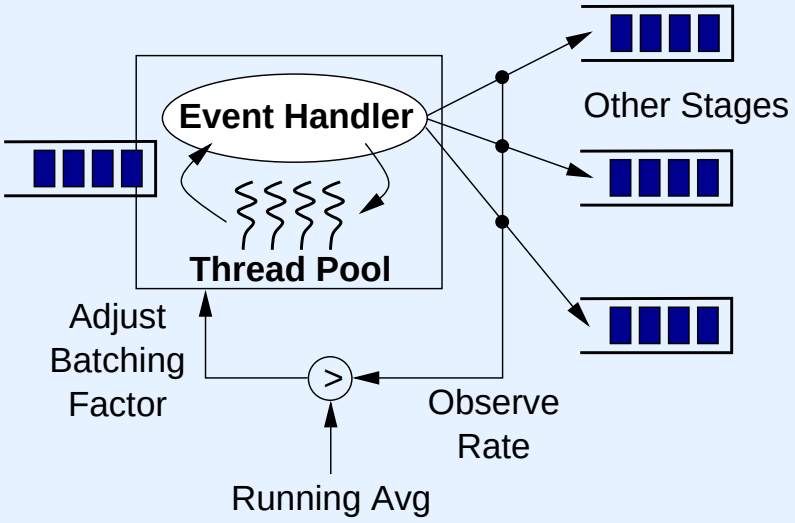
Goal: *Determine ideal degree of concurrency for a stage*

- Dynamically adjust number of threads allocated to each stage

Controller operation

- Observes input queue length, adds threads if over threshold
- Idle threads removed from pool

SEDA Batching Controller



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

- **Batching factor:** number of events processed by stage at once
- Small batching factor $\longrightarrow$ low response time
- Large batching factor $\longrightarrow$ high throughput

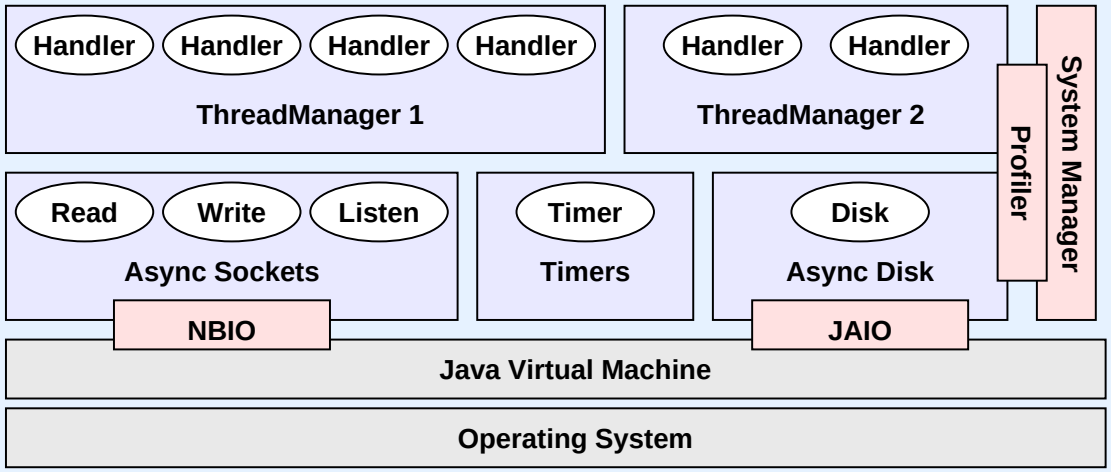
Attempt to find smallest batching factor with stable throughput

- Observes rate of events outgoing from stage
- Reduces batching factor when throughput high, increases when low

Outline of this Talk

- Problems with Existing Concurrency Models
- The SEDA Architecture
- Dynamic Resource Controllers
- SEDA Asynchronous I/O Primitives
- Application Evaluation: HTTP and Gnutella Servers
- Future Work and Conclusions

SEDA Prototype: Sandstorm



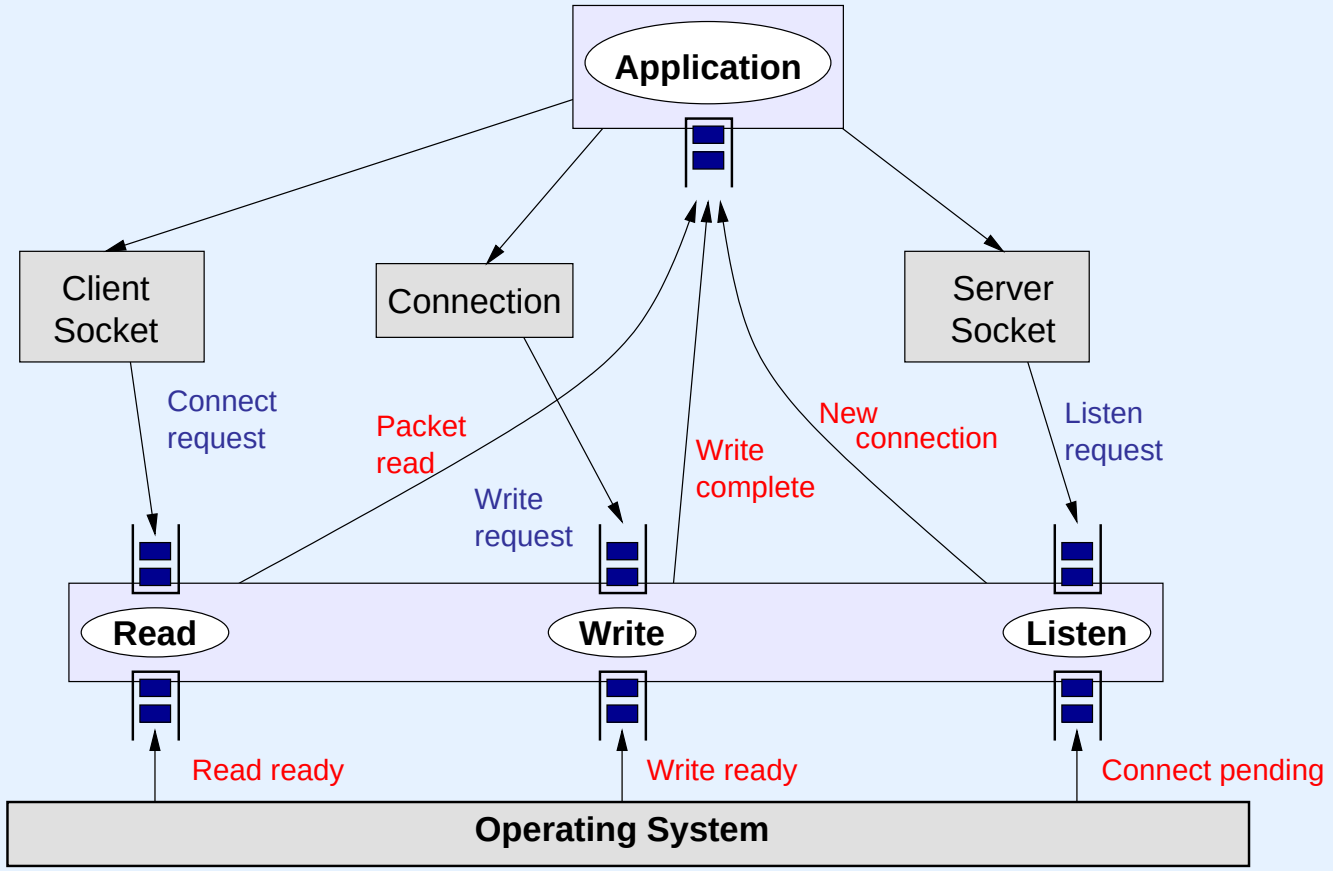
Implemented in Java with nonblocking I/O interfaces

- NBIO: Nonblocking socket I/O and `poll()` for Java
- JAIO: Nonblocking disk I/O via POSIX.4 AIO *(under construction)*

Java pros and cons

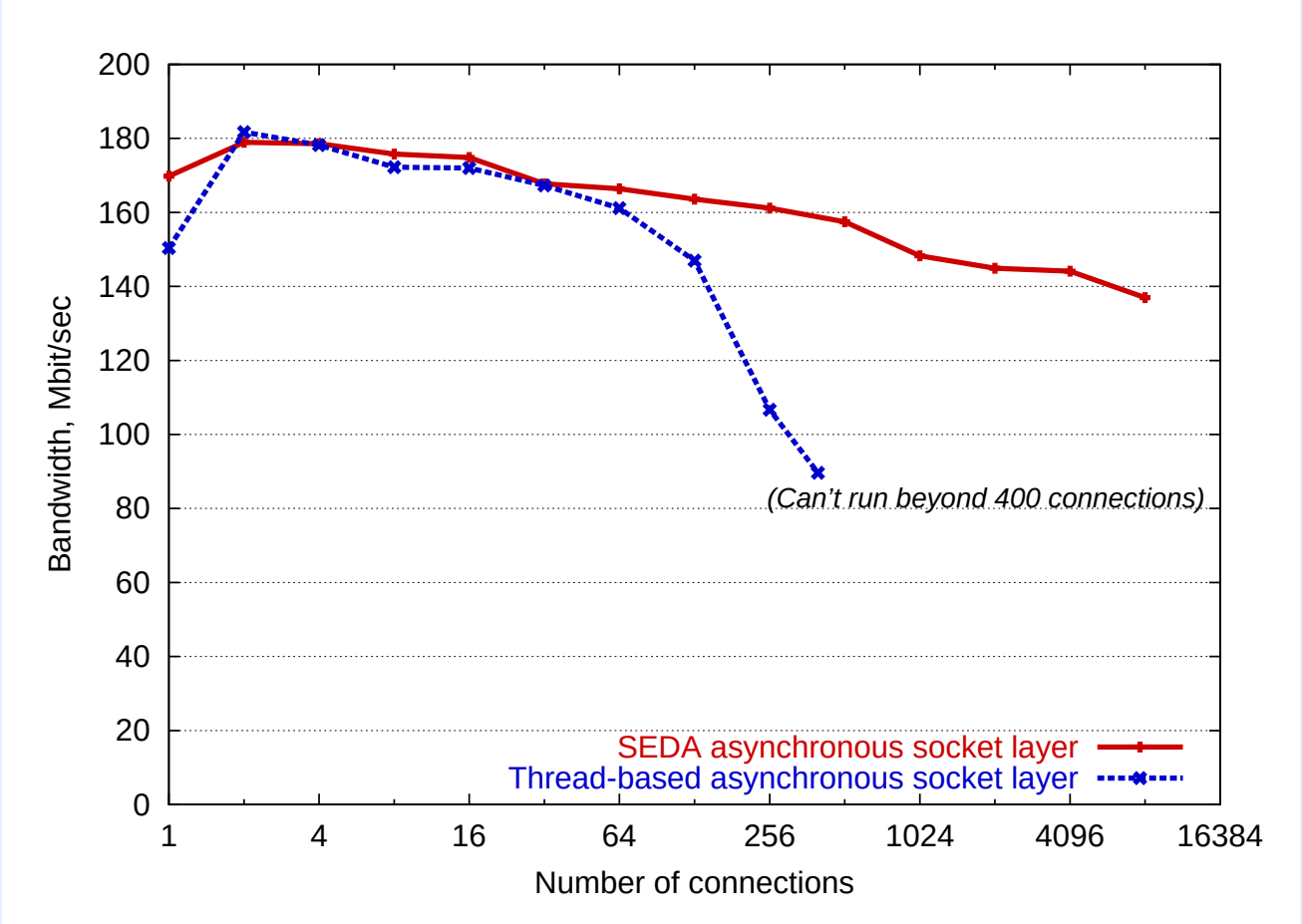
- Automatic memory management greatly simplifies design
  - ▷ *No need to track event propagation and usage*
- Performance hit non-negligible, but latest compilers are closing the gap

SEDA Asynchronous Sockets Layer



- Makes use of nonblocking I/O provided by O/S
- Each stage responds to user and O/S events (from two queues)
  - ▷ *Designed to support many thousands of connections*
  - ▷ *Non-trivial scheduling issues internally!*

Asynchronous Sockets Performance



(4-way 500 MHz PIII, Gigabit Ethernet, Linux, IBM JDK 1.1.8)

- Server reads 1000 8kb packets, sends 32-byte ack
- Per-user thread limit of 512 exceeded in threaded case
- SEDA obtains *137 Mbps* for *8192* connections

Other Sandstorm Components

Asynchronous Disk Layer

- Uses *blocking* file I/O and thread pool for concurrency
- Size of thread pool managed by SEDA stage controller
  - ▷ *Dynamically adjust size of thread pool based on perceived demand*
- On the horizon: true asynchronous implementation using POSIX.4 AIO

Timers

- Special-purpose stage which fires events at requested intervals

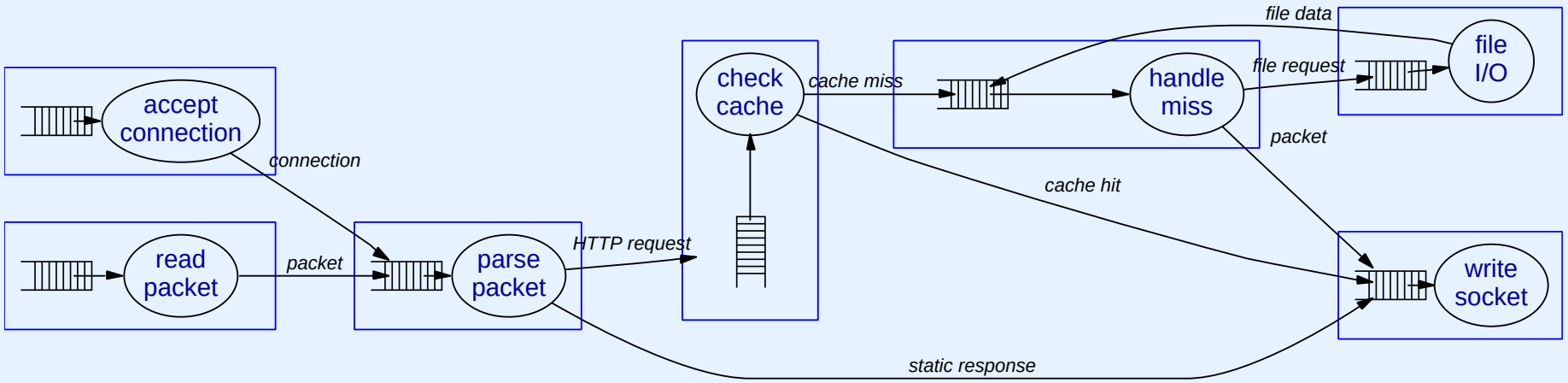
System Manager Interface and Profiler

- Management functions: Obtain stage handles, create/destroy stages, etc.
- Profiler: generate temporal trace of event queue lengths
- Automatic visualization of stage connectivity

Outline of this Talk

- Problems with Existing Concurrency Models
- The SEDA Architecture
- Dynamic Resource Controllers
- SEDA Asynchronous I/O Primitives
- Application Evaluation: HTTP and Gnutella Servers
- Future Work and Conclusions

Haboob: A SEDA-Based Web Server



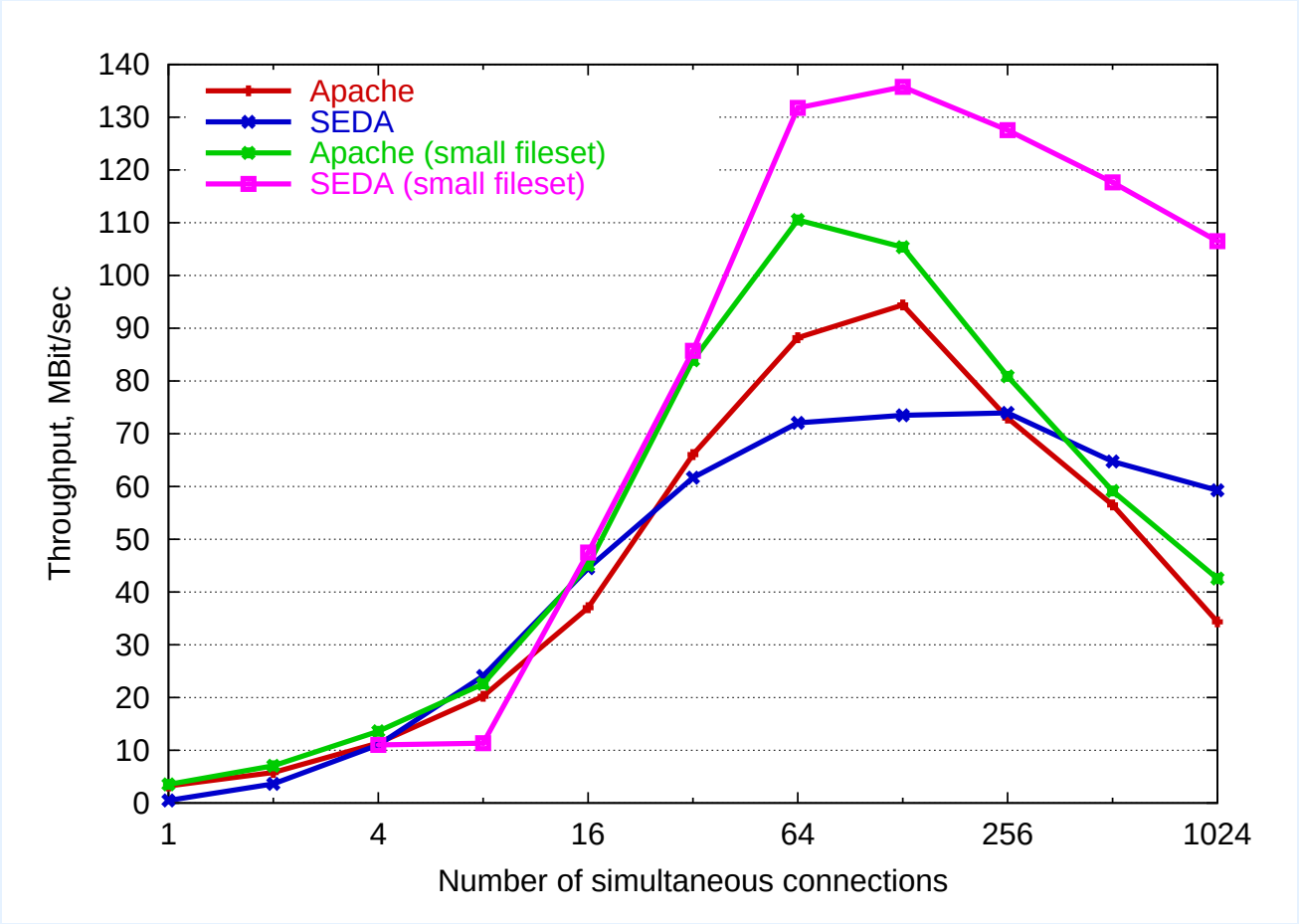
Supports *static file load* from SpecWEB99 benchmark

- ''Small'' fileset: 139 MB, ''Large'' fileset: 3.31 GB
- Page size ranges from 102 Bytes to 940 KB
- Clients sleep 20 msec between requests, 5 requests/connection

Manages in-memory cache of recently accessed pages

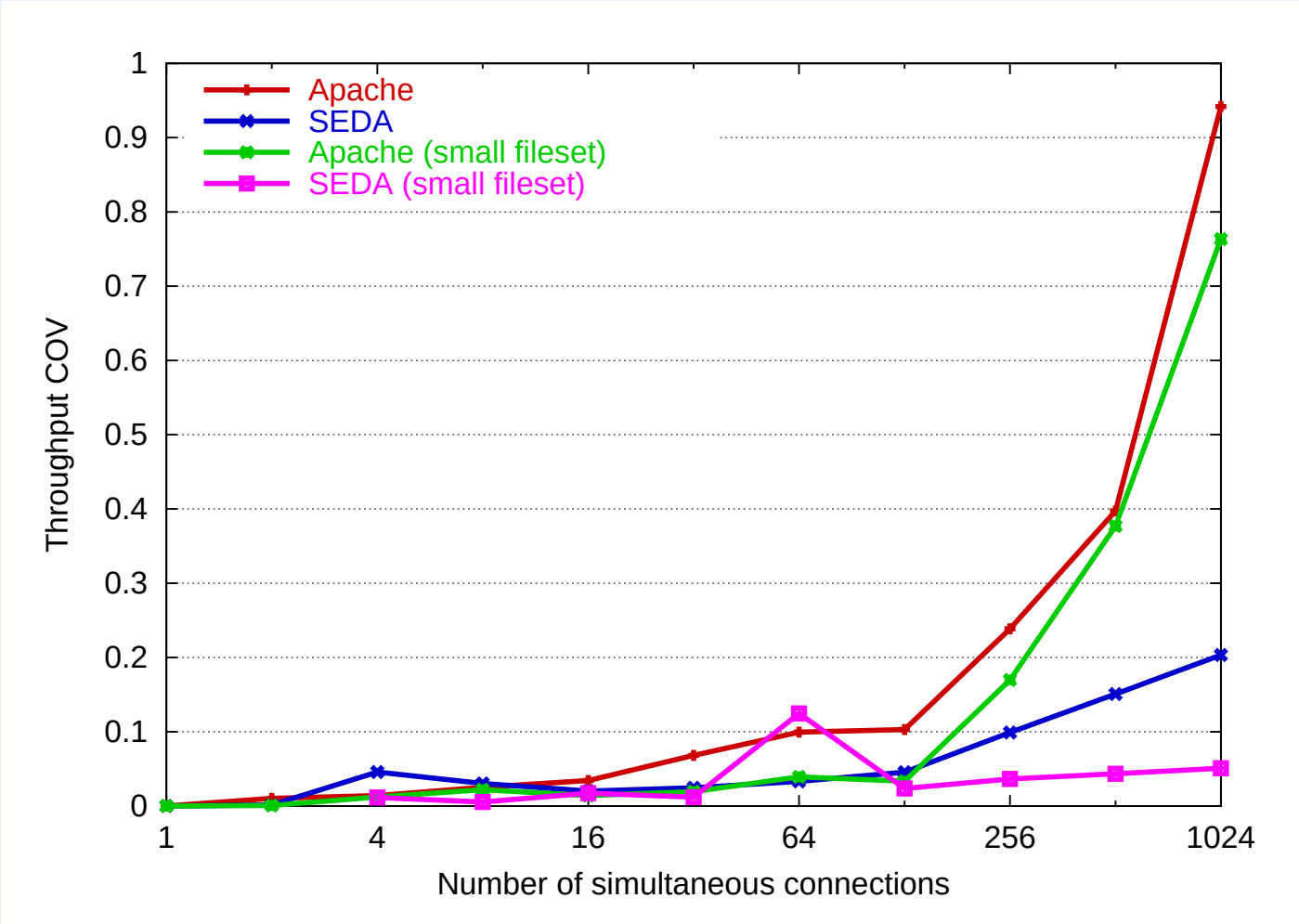
- Cache size of 200 MB
- Uses Shortest Connection First (SCF) scheduling

Haboob Throughput vs. Apache



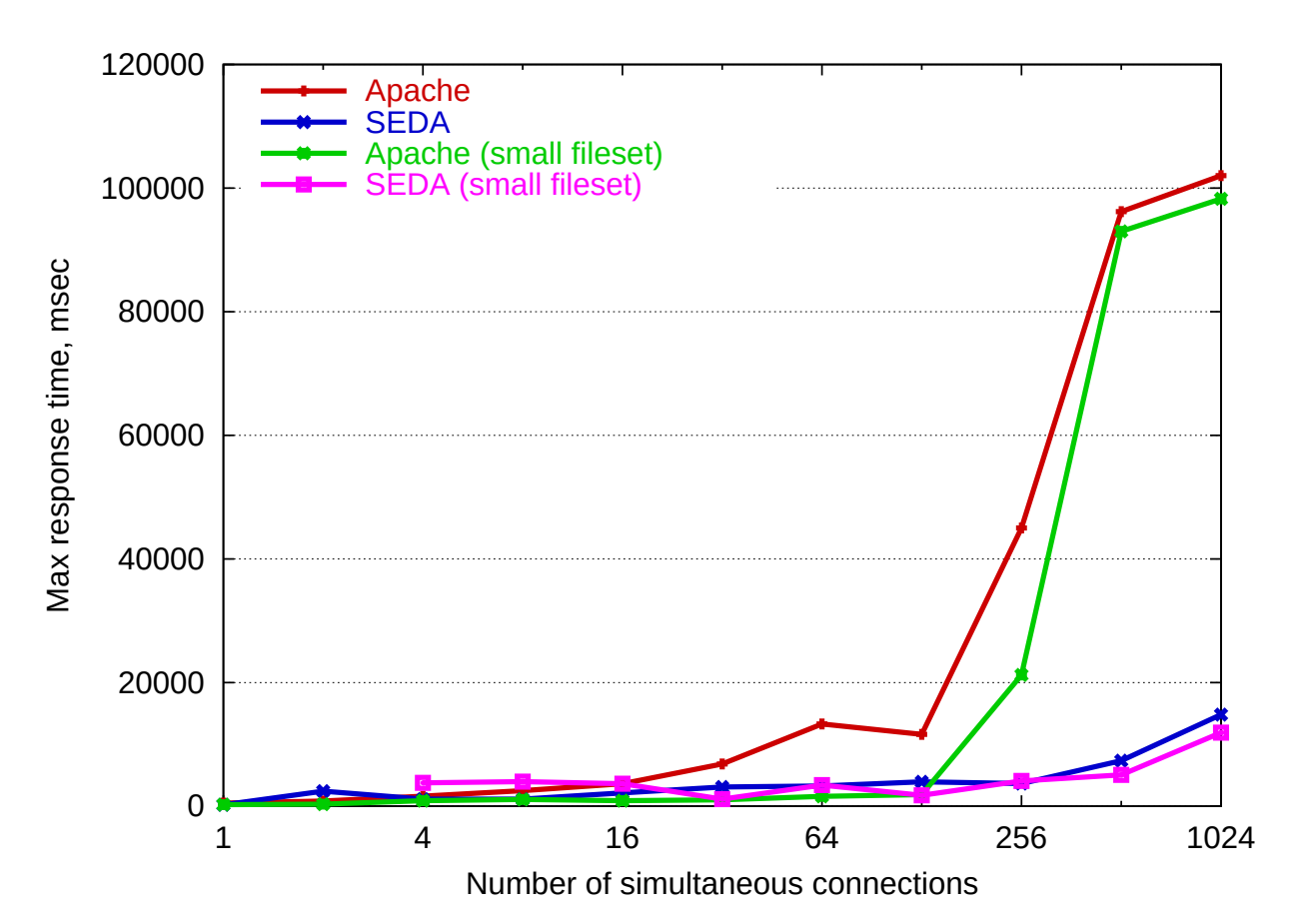
- SEDA performance stable for large number of connections
  - ▷ *Some degradation due to Linux socket inefficiencies*
- Apache degrades noticeably

Haboob Fairness vs. Apache



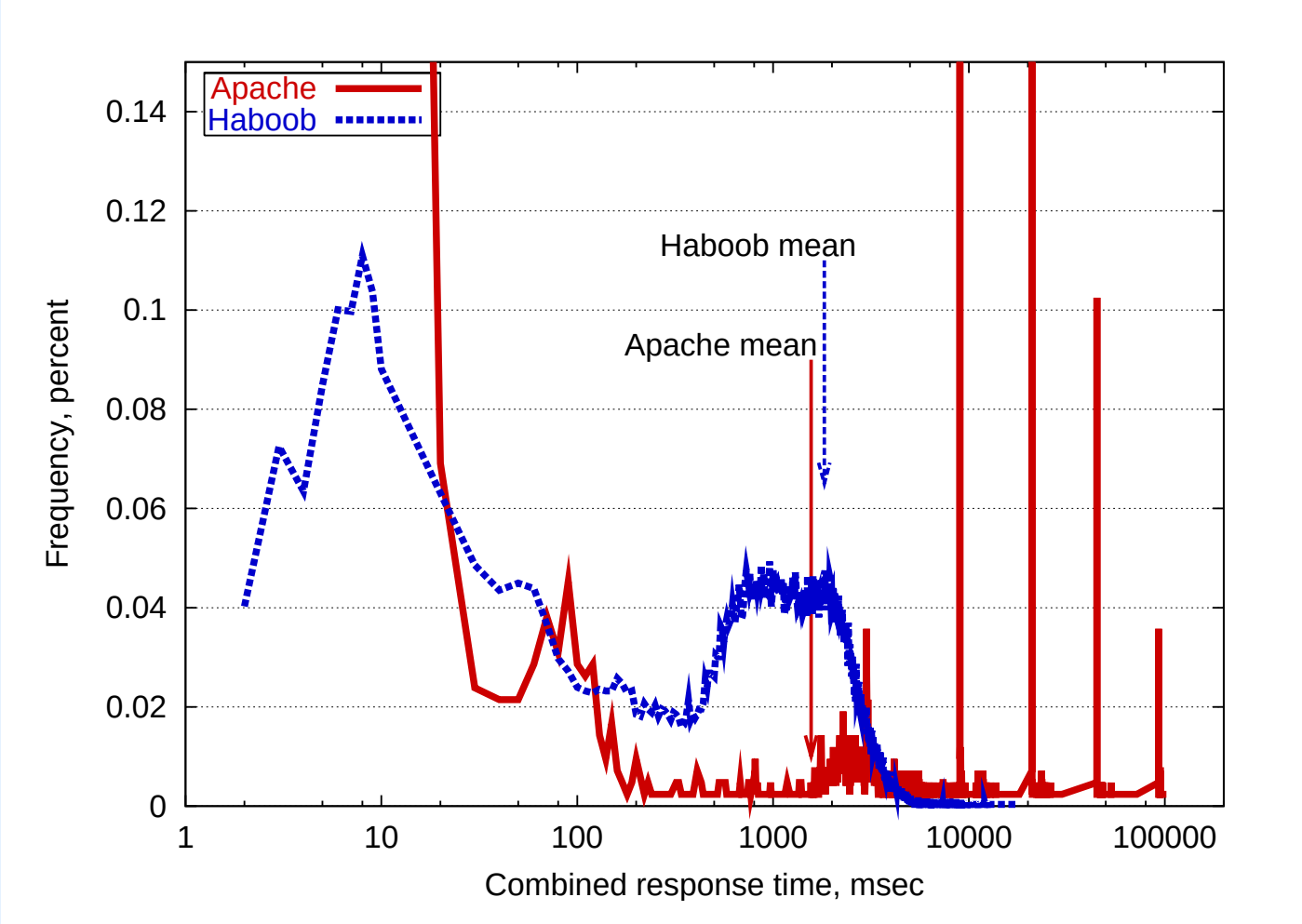
- Coefficient of variation: *standard dev / mean*
- Low COV $\rightarrow$ high degree of fairness to clients

Haboob Maximum Response Time vs. Apache



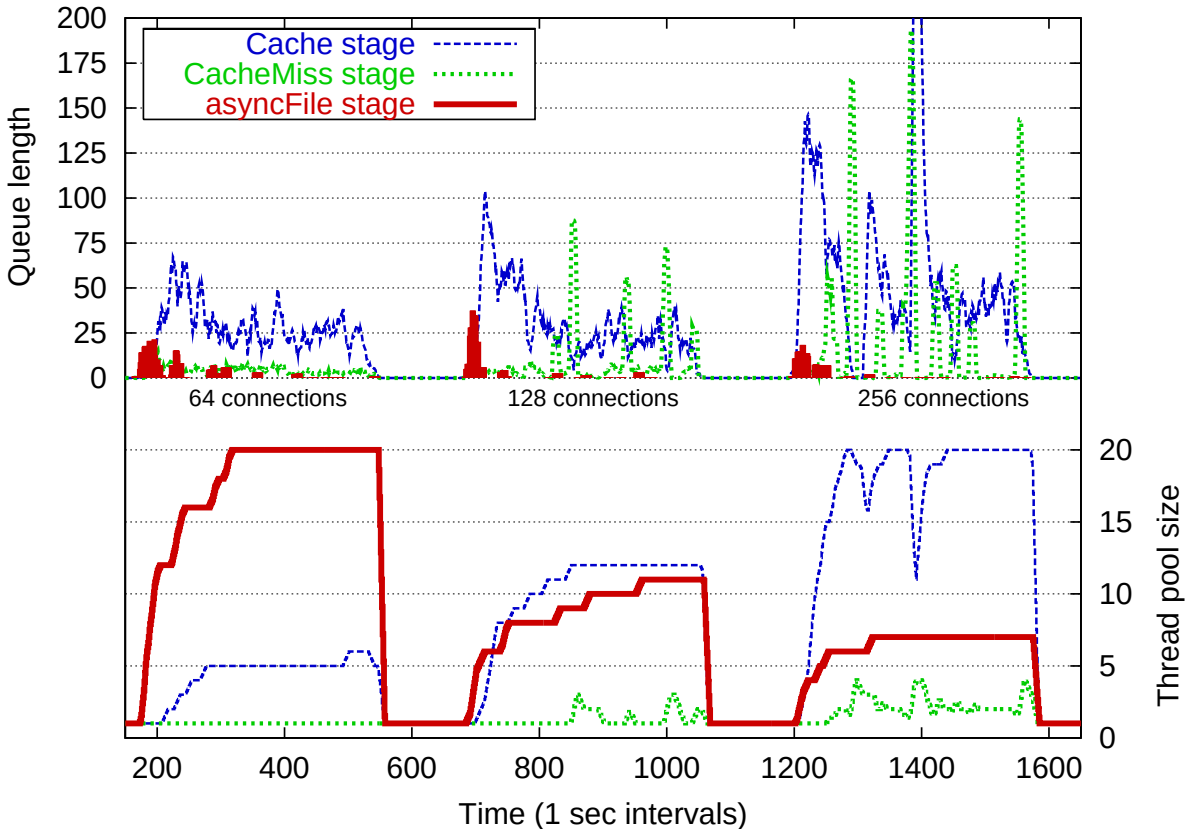
- SEDA exhibits low maximum response times
- Apache max response time is unbounded as load exceeds capacity
  - ▷ *Client must wait for a server process to become available*
  - ▷ *Exacerbated by exponential backoff in TCP retransmission timer*

Response Time Histogram



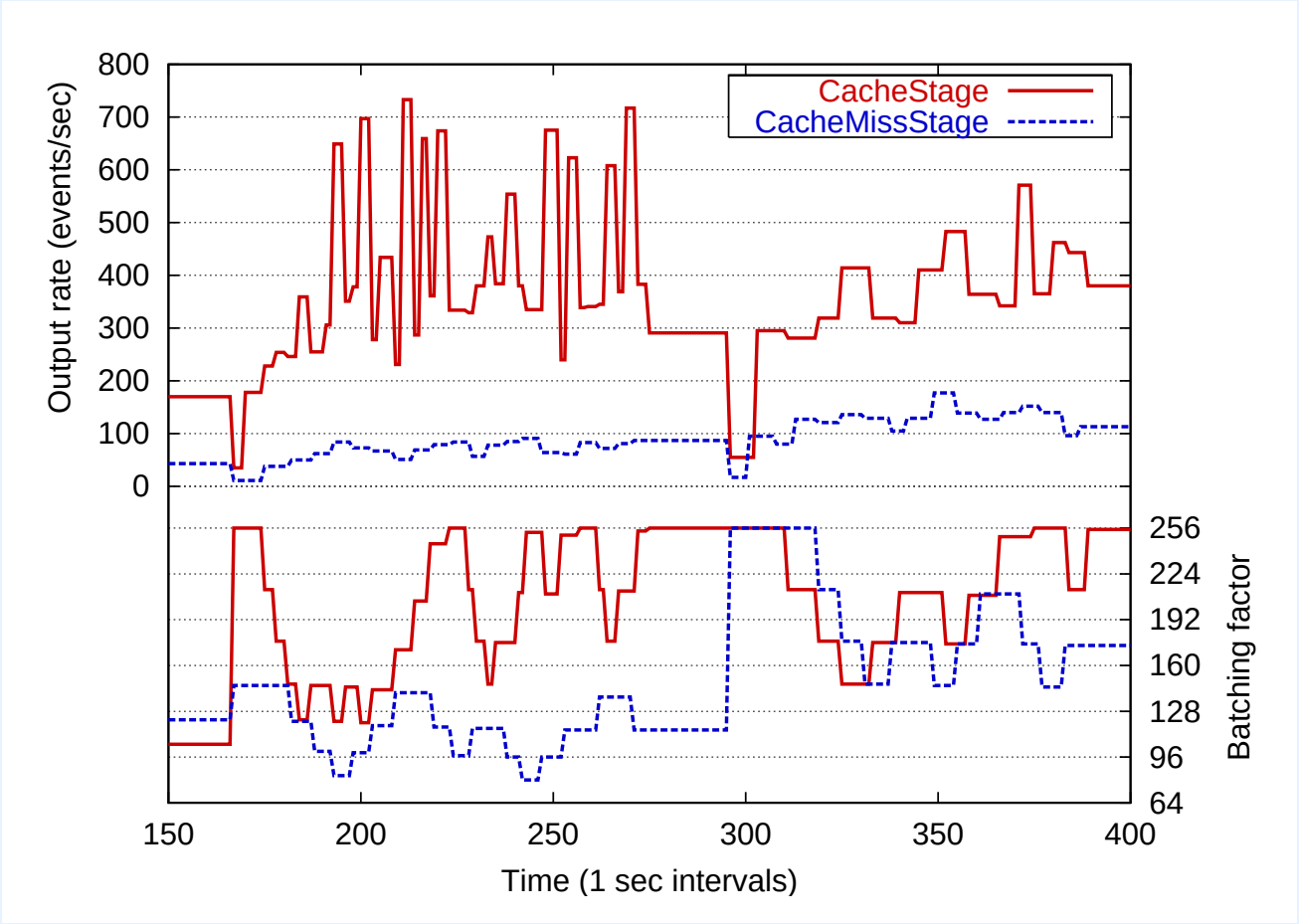
- SEDA: mean *1.8 sec*, max *14 sec*
- Apache: mean *1.5 sec*, max *1.7 minutes*
  - ▷ *Large spikes in Apache due to TCP retransmit timer backoff*

Thread Pool Controller Operation



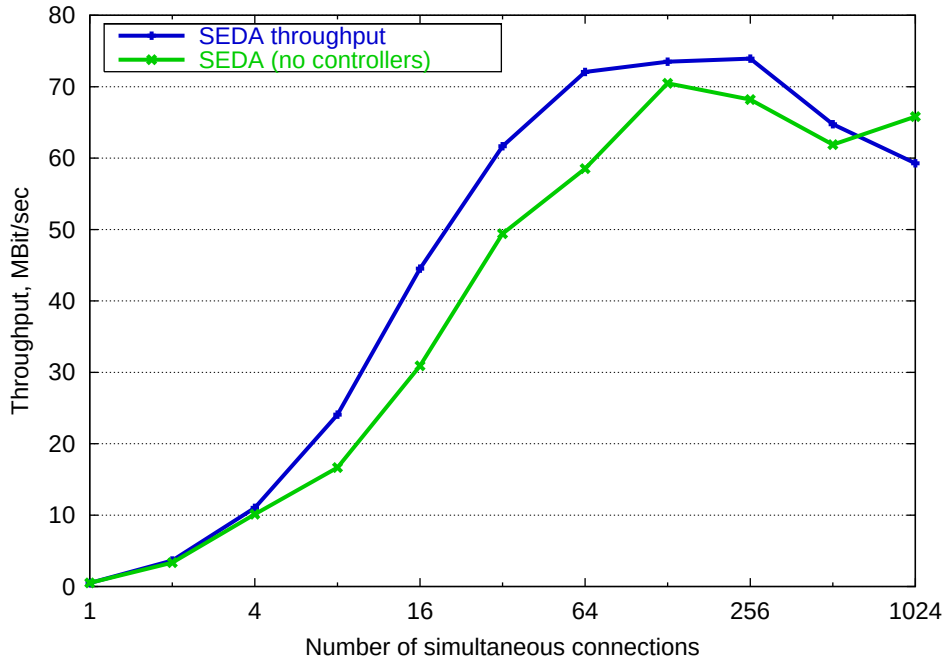
- Adds thread to stage when queue reaches threshold
  - ▷ *Queue threshold of 100 entries, max threads 20 per stage*
- Fewer threads needed for file I/O over time
  - ▷ *Due to filesystem buffer cache warming up*

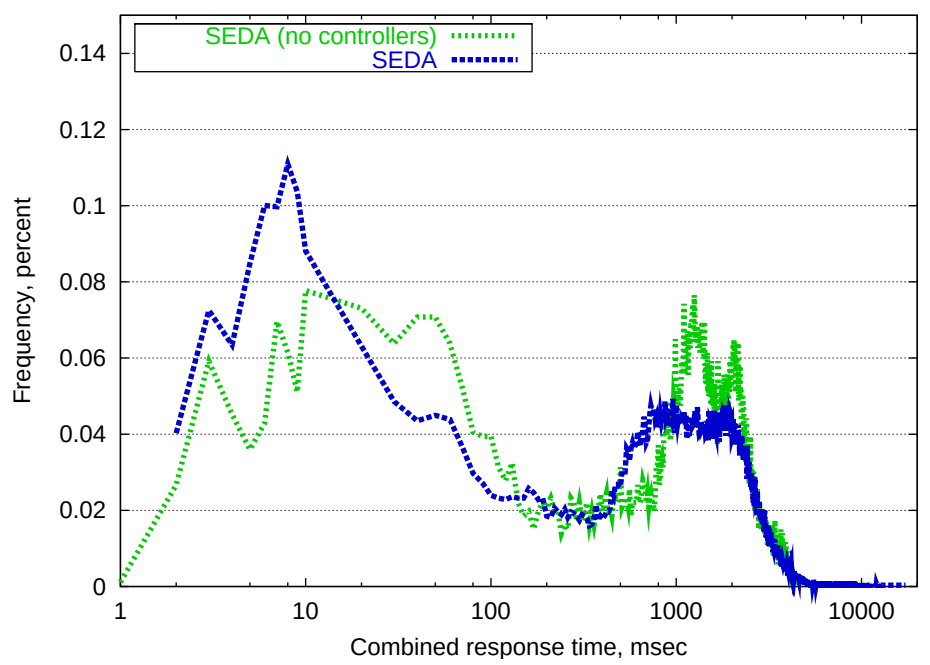
Batching Controller Operation



- Reduce batching factor while output event rate is stable
- At light load, maximum batching $\longrightarrow$ high throughput, low inherent response time
  - ▷ *Respond to sudden drop in load by resetting batching factor to max*

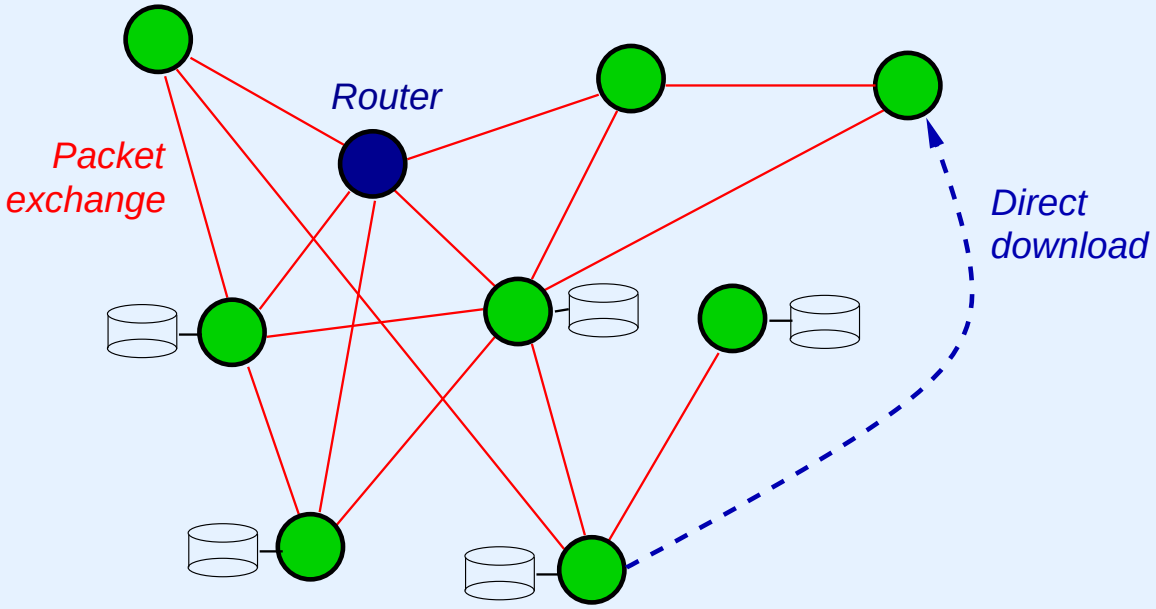
Performance Effect of Controllers





- Thread pool controller leads to faster throughput ramp-up
- Batching controller leads to reduced response time
- More investigation of these effects underway

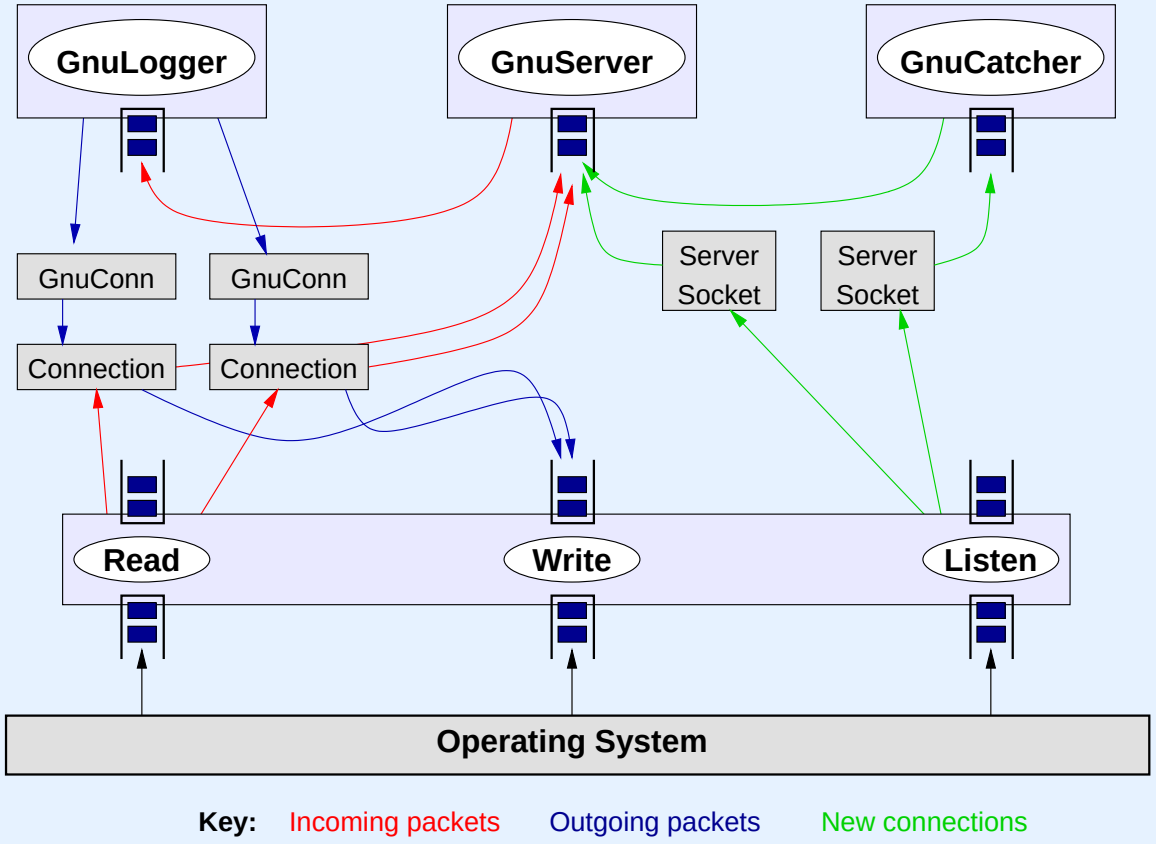
Gnutella Packet Router



Gnutella basics

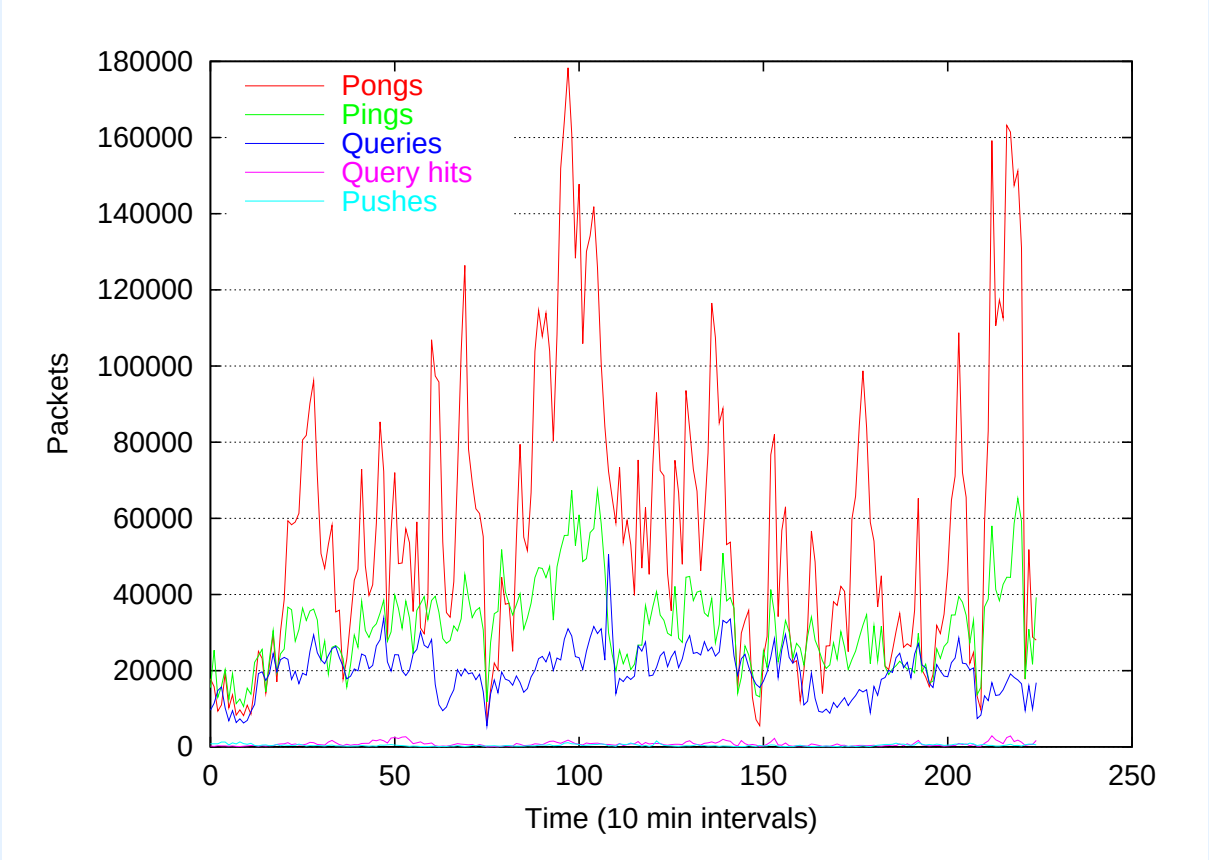
- Decentralized peer-to-peer file sharing network
- Every node exchanges messages with its neighbors
  - ▷ *ping, pong, query, queryhit, push message types*
- Direct download from host via HTTP
- Initial discovery via well-known host
- Several thousand users at any time, 10's of TBs of data

Gnutella Router Structure



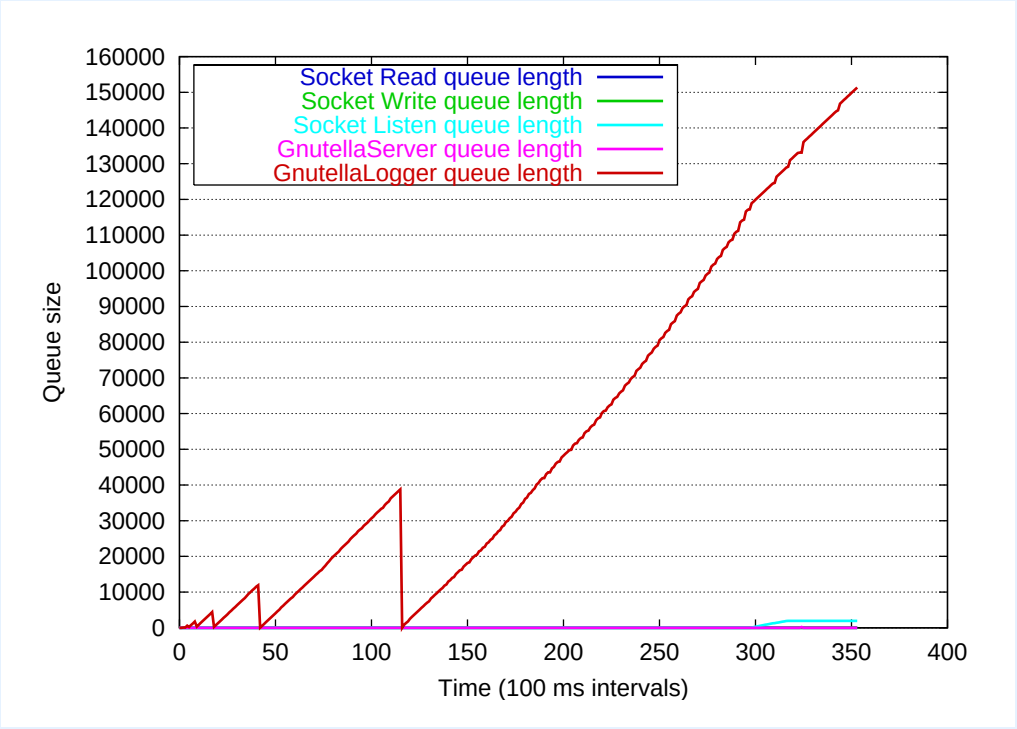
- Logger stage: Routes and logs packets
- Server stage: Parses incoming packets
- Catcher stage: Establishes new connections
- Gnutella Connection: Formats outgoing packets

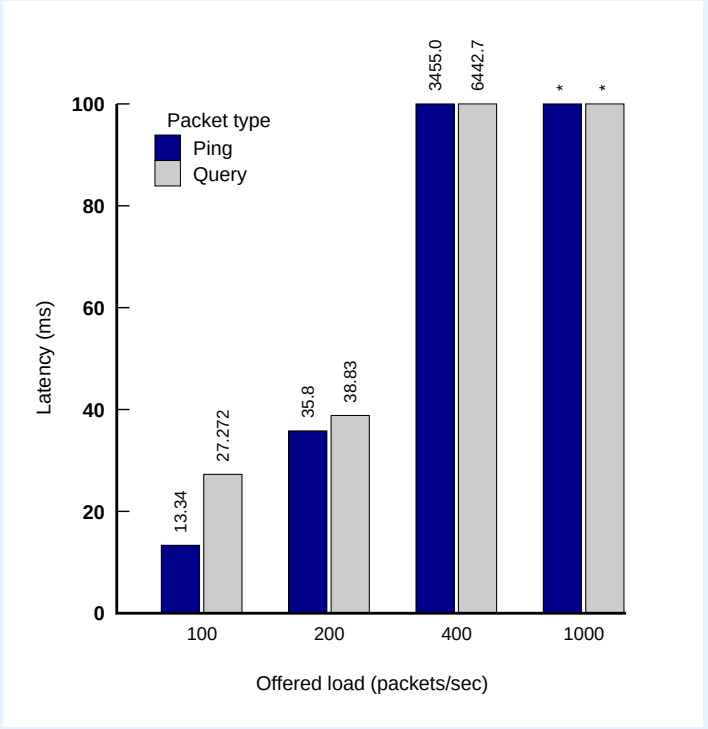
Gnutella Packet Trace



- Gathered over 37-hour period
- *24.8 million* packets, average *179.55* per sec
- *72396* connections, average 12 at any time
- Very bursty, no clear diurnal pattern

Router Latency Under Overload





- Benchmark client generates realistic packet streams
- Introduce intentional bottleneck into server:
  - *Server-side delay of 20 ms for query messages (15% of traffic)*

Dealing with Overload

Event queue thresholding

- Works, but drops many packets

Event queue filtering/reordering

- Allow non-query packets; drop query packets at threshold
- Service query packets last

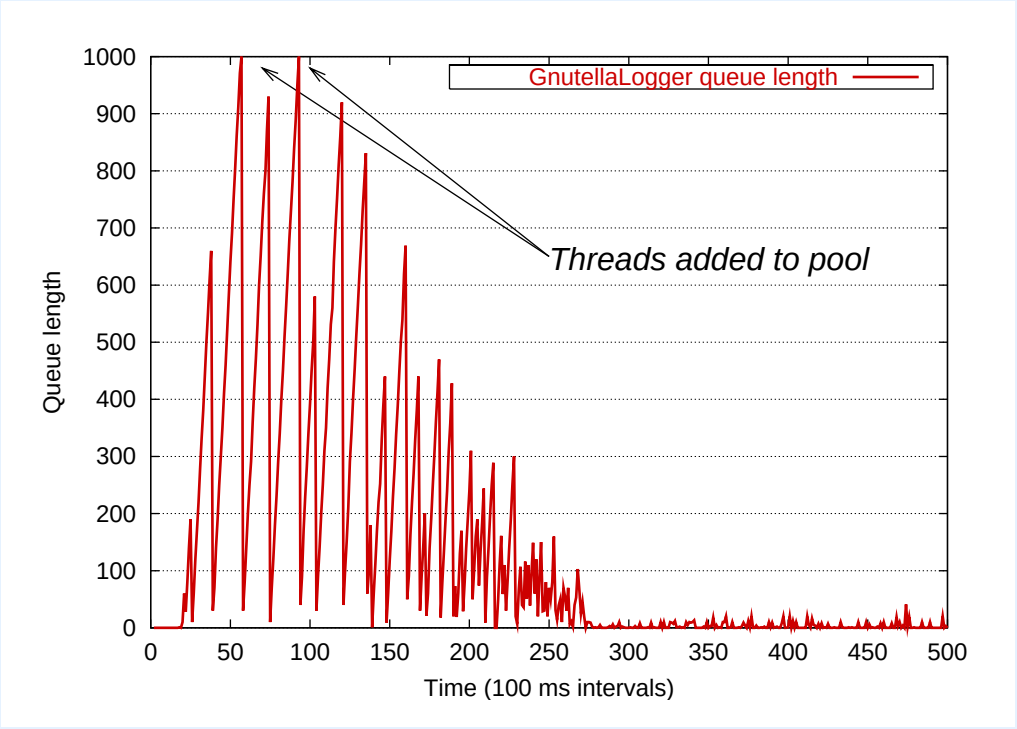
Thread pool resizing

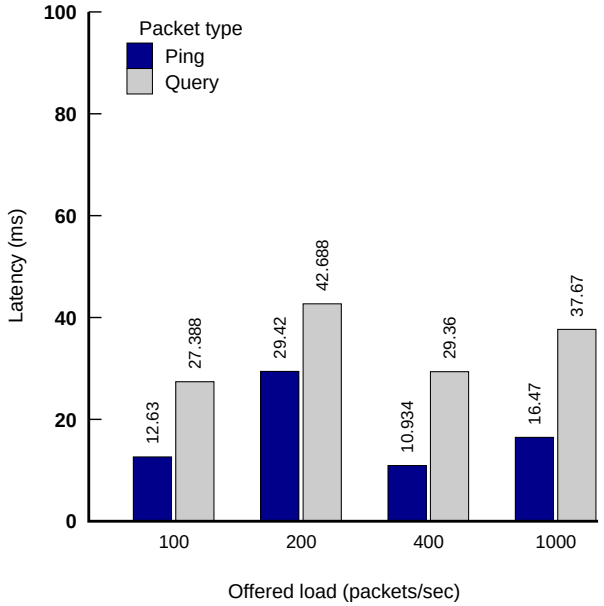
- Devote more threads to bottleneck stage
- Model stage as G/G/n queueing system
- n threads, arrival rate λ, query frequency p, query servicing delay L

$$n = \lambda p L = (1000)(0.15)(20 \text{ ms}) = 3 \text{ threads}$$

  - ▷ *Unfortunately, can't determine a priori*
  - ▷ *Instead, managed by SEDA thread pool controller*

Thread Pool Controller in Action





- Dynamically adjust size of thread pool for each stage
  - *Sample queue lengths every 2 sec*
  - *Add a thread when queue reaches threshold*
- 2 threads added to *GnutellaLogger* stage
  - *Matches theoretical result*

Related Work

High-performance Web servers

- Many systems realizing the benefit of event-driven design
- *[Flash, Harvest, Squid, JAWS, ...]*
- Engineered for specific application rather
- Little work on load conditioning, event scheduling

StagedServer (Microsoft Research)

- Core design similar to SEDA
- Primarily concerned with cache locality
- Wavefront thread scheduler: last in, first out

Click Modular Router, Scout OS, Utah Janos

- Various systems making use of structured event queues
- Packet processing decomposed as stages
- Threads call through multiple stages
- Major goal is latency reduction

Related Work 2

Resource Containers *[Banga]*

- Similar to Scout ''path'' and Janos ''flow''
- Vertical resource management for data flows
- SEDA applies resource management at per-stage level

Scalable I/O and Event Delivery

- *[ASHs, IO-Lite, fbufs, /dev/poll, FreeBSD kqueue, NT completion ports]*
- Structure I/O system to scale with number of clients
- We build on this work

Large body of work on scheduling

- Interesting thread/event/task scheduling results
- e.g., Use of SRPT and SCF scheduling in Web servers *[Crovella, Harchol-Balter]*
- Alternate performance metrics *[Bender]*
- We plan to investigate their use within SEDA

Future Work

Generalize load-conditioning mechanisms

- Credit-based flow control between stages
  - ▷ *More expensive tasks require more credits*
- Extend resource control to memory, other resources?
- Experiment with alternate schedulers

Develop control-theoretic approach to resource management

- Large body of prior work in control of physical systems
- How much of it can be applied here?

Evaluate with other ''challenge'' applications

- Dynamic Web server content w/ general-purpose scripting
- Gnutella network crawler & search engine
- Incorporation into OceanStore, Telegraph, Ninja projects

New Way of Thinking about Software

Support for massive concurrency requires new design techniques

- SEDA introduces service design as a *network of stages*
- Design for robustness and adaptivity, rather than best case
- Expose request streams to applications for load conditioning

Resource throttling to keep stages within operating regime

- Adapt behavior at runtime to deal with changing load
- Controllers shield service developers from much of this complexity

Implications for OS and language design

- SEDA model opens up new questions in service design space
- Bring body of work on control systems to bear on service design
- Many interesting controller algorithms possible

Summary

Staged Event-Driven Architecture designed to support:

- Massive concurrency -- 10000's of connections
- Robust performance under wide variation in load
- General-purpose toolkit for Internet service programming

Many interesting research directions

- Scheduling and resource management
- Load conditioning policies
- Dynamic resource controllers

Promising initial results

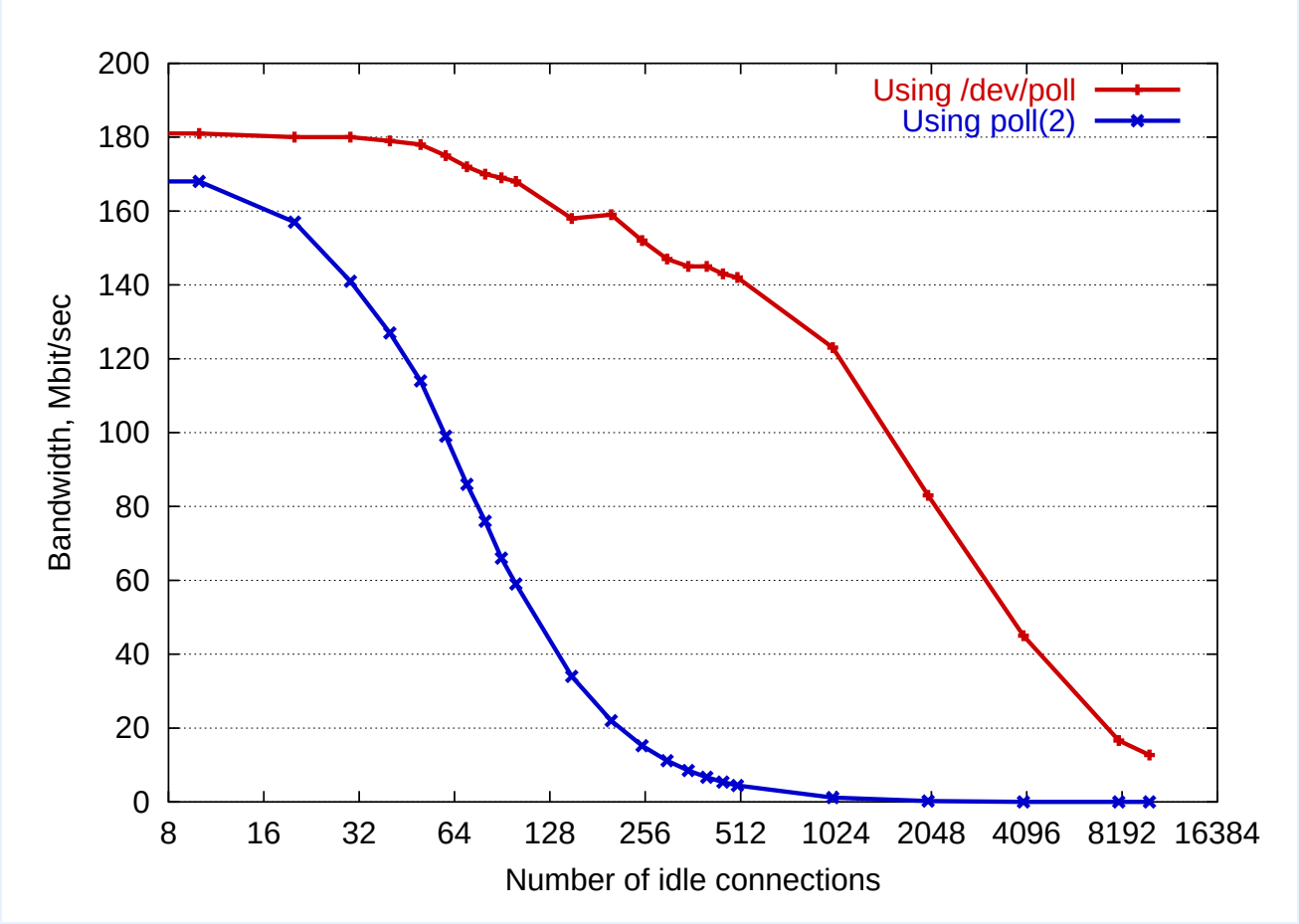
- *Sandstorm* service platform
- Demonstrated application scalability and load conditioning

For more information

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

Backup Slides Follow

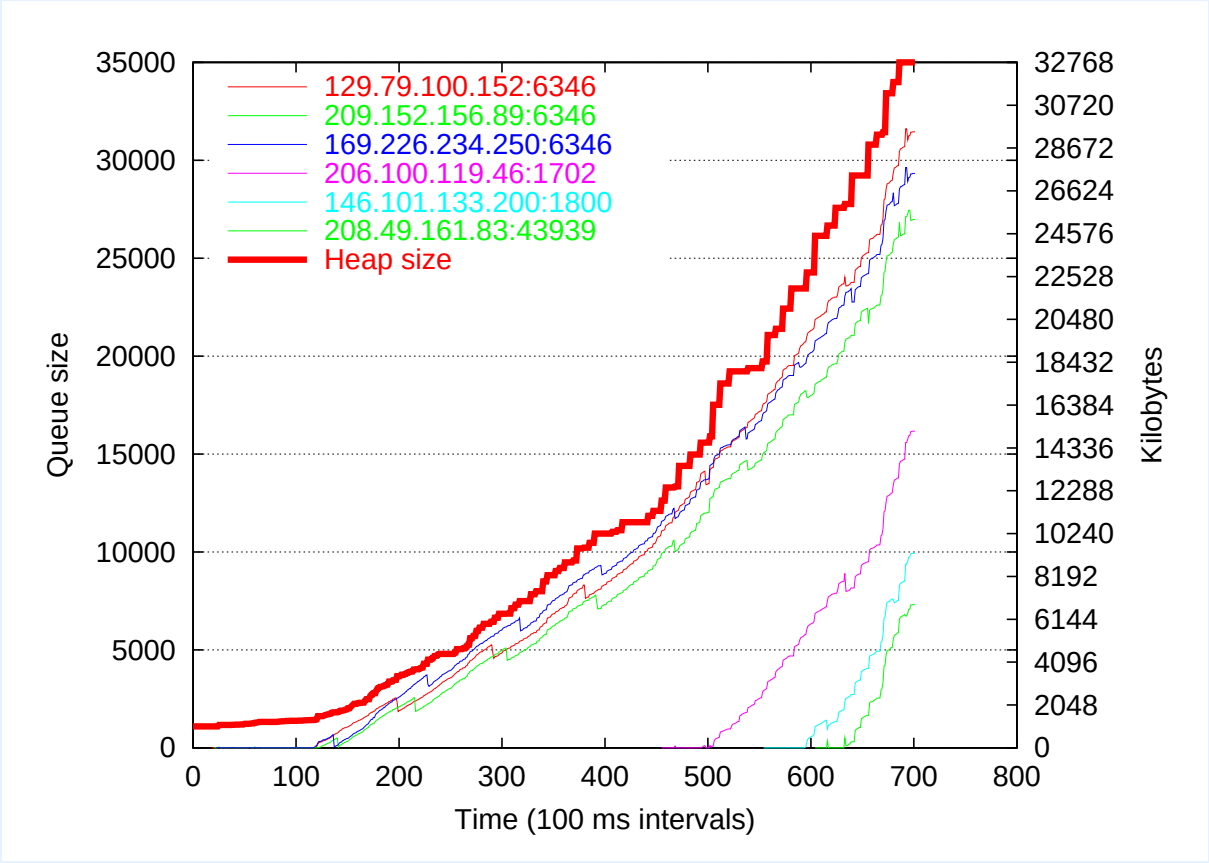
Effect of Idle Connections



(4-way 500 MHz PIII, Gigabit Ethernet, Linux, IBM JDK 1.1.8)

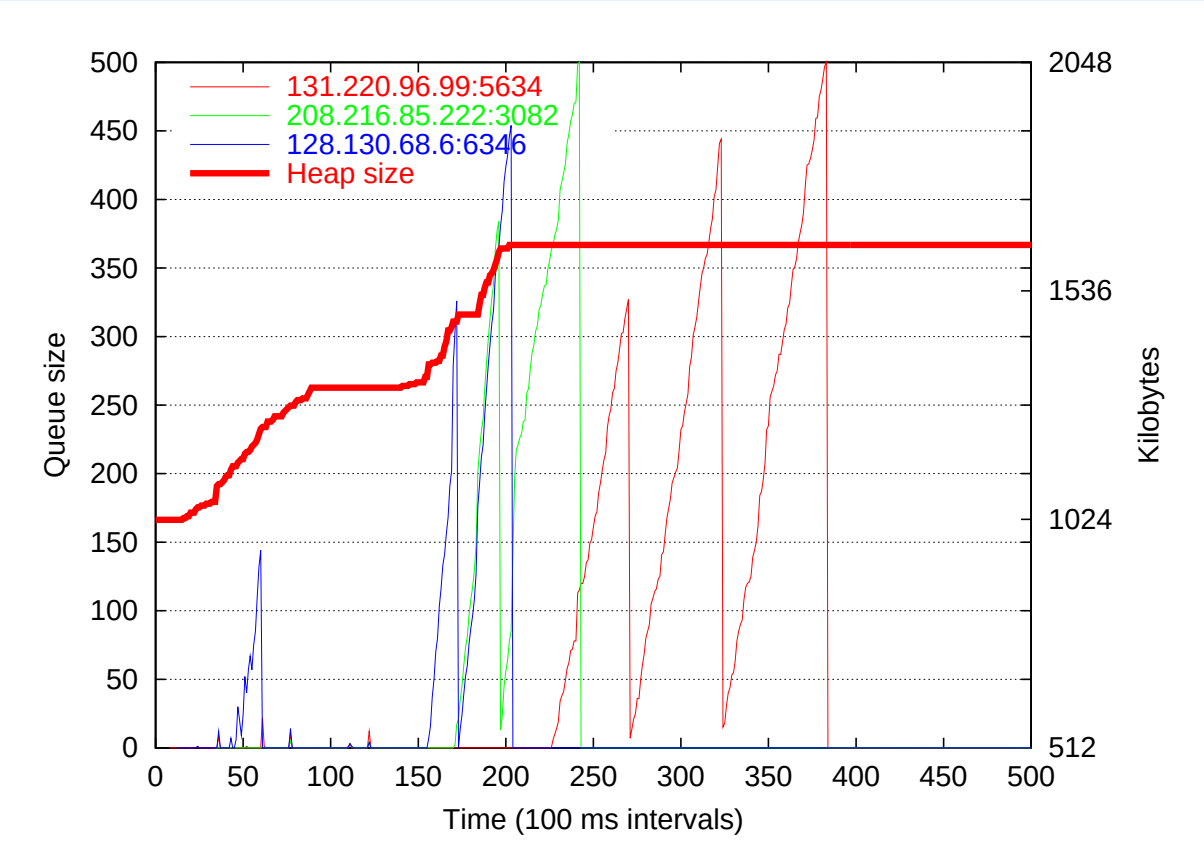
- One active connection, 1-10000 idle connections
- Compare poll(2) to /dev/poll event dispatch

Dealing with Clogged Connections



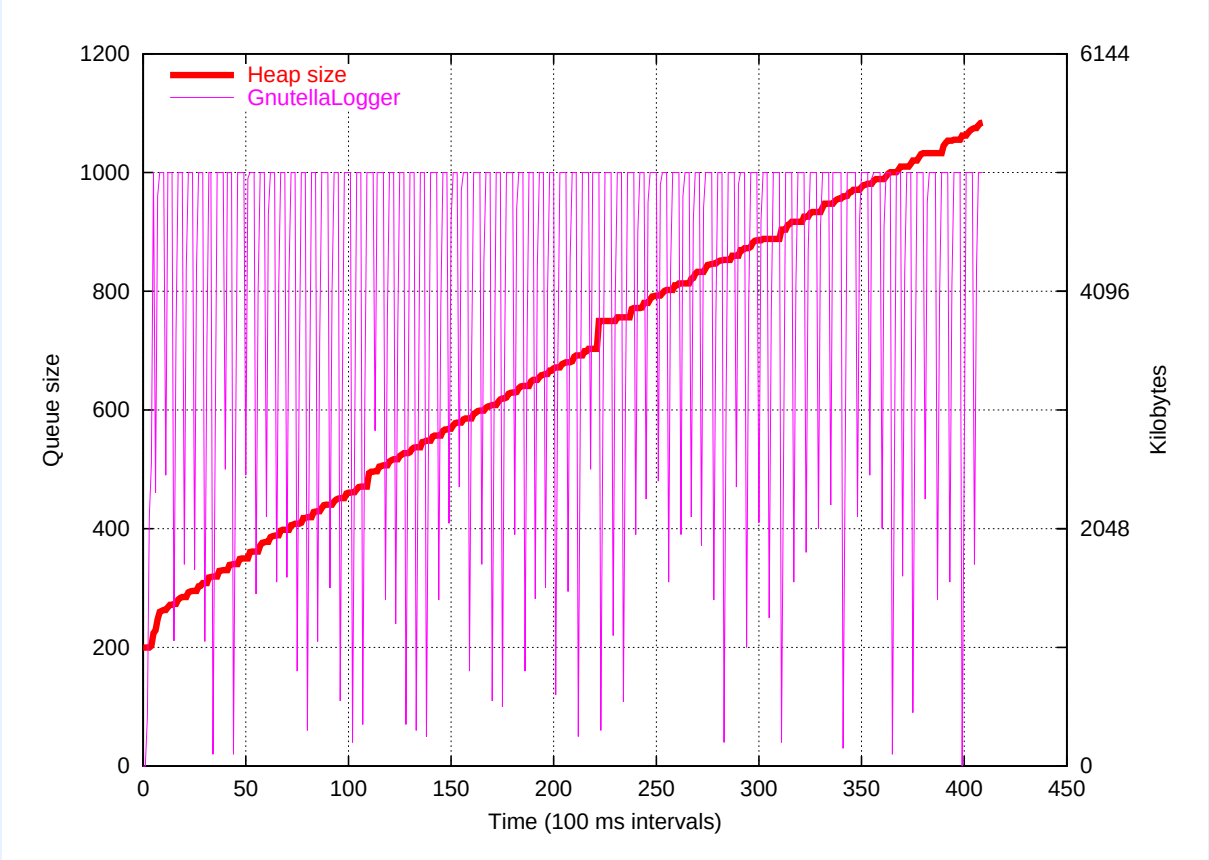
- Server would crash after a few hours
- Cause: saturated connections
  - ▷ *115 packets/sec can saturate a 28.8 modem link*

Socket Queue Thresholding



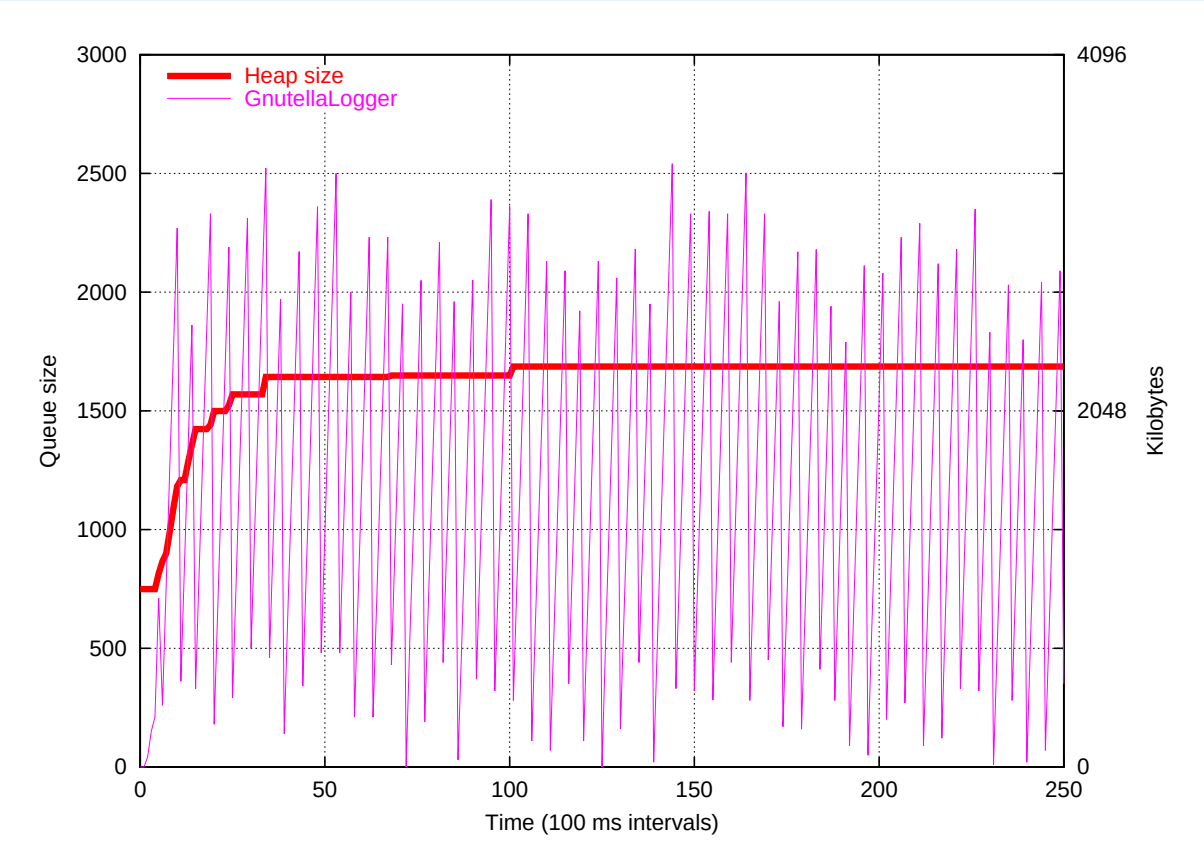
- Close connection if outgoing queue reaches threshold
- One form of load conditioning
- Many variations possible

Event Queue Thresholding



- Threshold incoming event queue at 1000 entries
- Heap size continues to grow! Why?
  - ▷ *Gnutella server maintains list of recent packets*
  - ▷ *Timer event used to clean out list, but is being dropped*

Application-Specific Event Filtering



- No queue threshold; Gnutella server does its own filtering
- Threshold only the number of query packets processed
- All other events processed normally