The Staged Event-Driven Architecture for Highly Concurrent Servers

Matt Welsh

mdw@cs.berkeley.edu

Ph.D. Qualifying Examination
December 13, 2000

Internet Services Today

Massive concurrency demands

- Yahoo: 780 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
- Service should not overcommit resources

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*

Problem Statement

Building highly concurrent applications is hard

- Existing software architectures not entirely adequate
- Few tools exist to help

Thread-based Concurrency

- Too heavyweight for massive scalability
- Designed for timesharing:
  - ▷ *O/S multiplexes ‘‘virtual machines’’ on hardware*
  - ▷ *Synchronization primitives are expensive*

Event-Driven Concurrency

- Not well-supported by O/S or languages
- Systems usually designed from scratch
  - ▷ *Code is rarely modular or reusable*
- Resource management is challenging
  - ▷ *Difficult to distinguish multiple I/O flows*
- Debugging is difficult

Hypothesis

Proposal: the Staged Event Driven Architecture (SEDA)

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

Simplify task of building concurrent applications

- Staged structure supports modularity and reuse
- Apps not responsible for thread management, event scheduling, or I/O

Enable load conditioning

- SEDA supports fine-grained, app-specific resource management
- Event queues allow prioritization or filtering during heavy load
- Global resource management possible without intervention of apps

Support wide range of applications

- Not just tuned for specific app (e.g., Web servers)
- General-purpose architecture for servers

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)

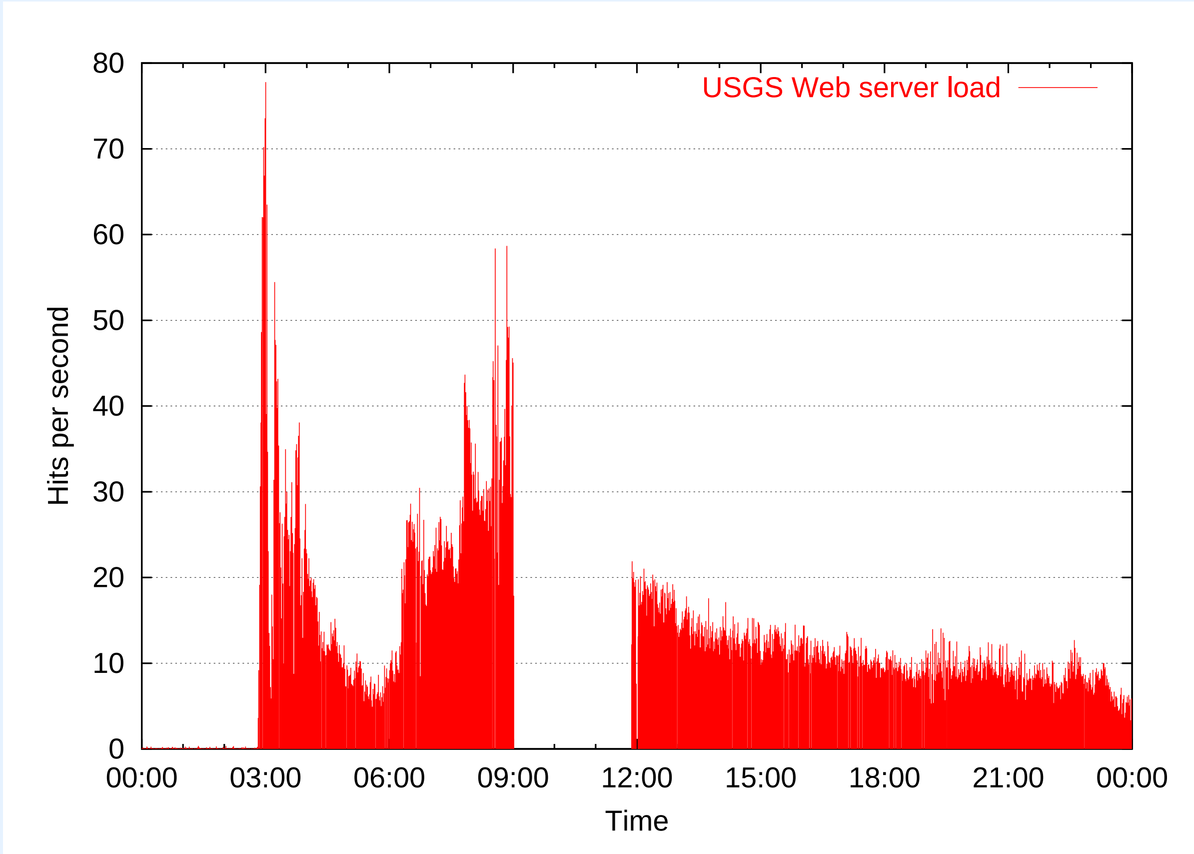
Overprovisioning is generally infeasible

- Peak demand many times that of average
- Service load is extremely bursty
  - ▷ *(cost of n machines) $>$ (n $\times$ cost of 1 machine)*

Must be well-conditioned to load

- Service should not overcommit resources
- Performance should not degrade such that all clients suffer

Load is Extremely Bursty



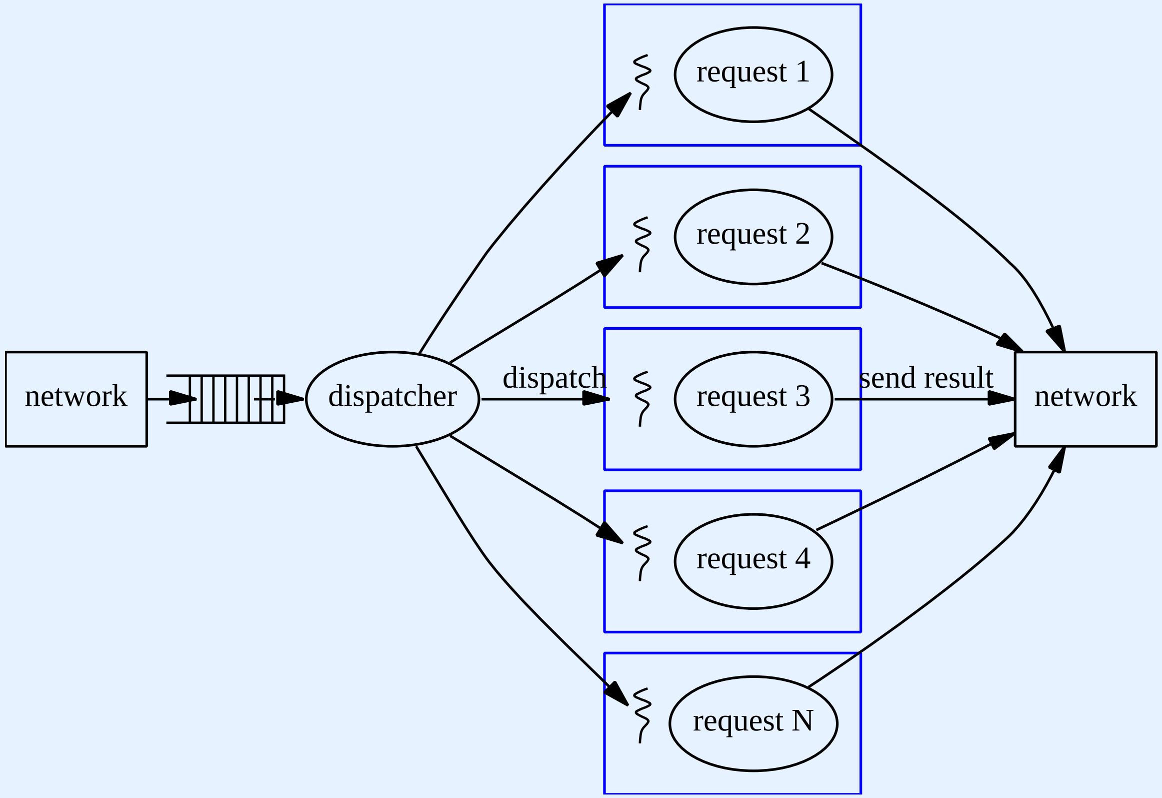
Web log from USGS Pasadena Field Office

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

Outline of this Talk

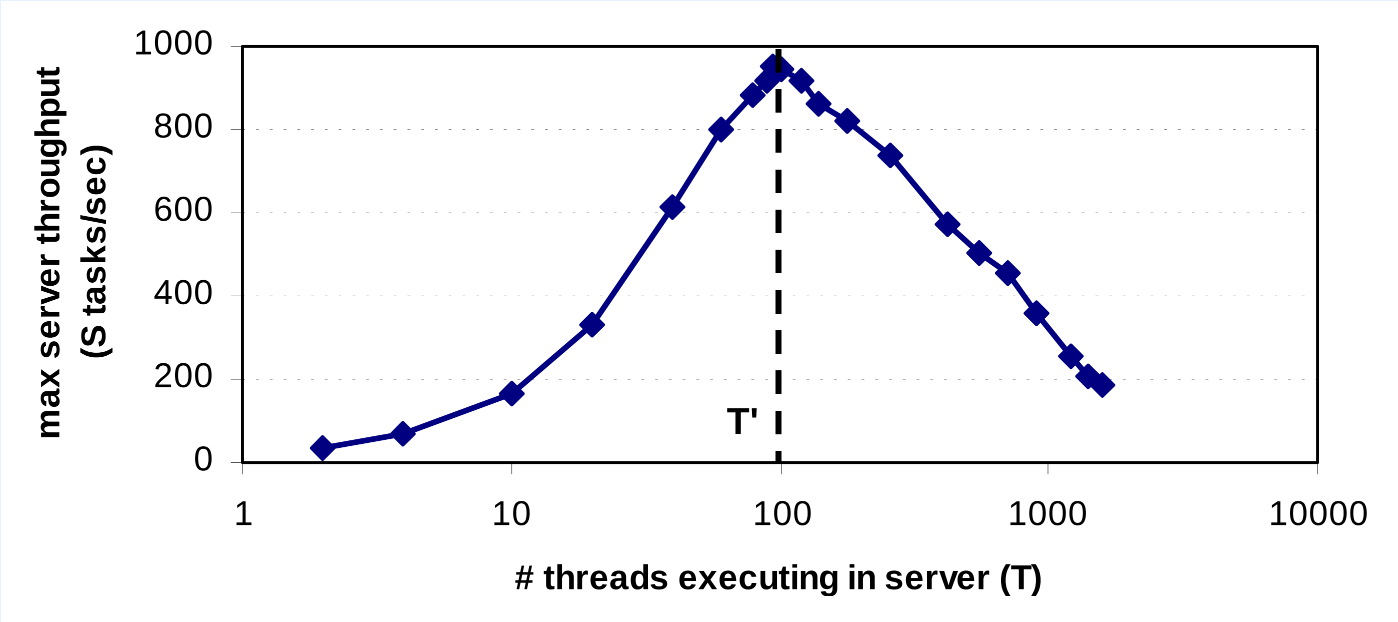
- Existing Concurrency Models
- SEDA Architecture Overview
- Research Issues
- Initial prototype: Sandstorm
- Application Evaluation: HTTP and Gnutella
- Related Work and Timeline

Thread-Based Concurrency



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

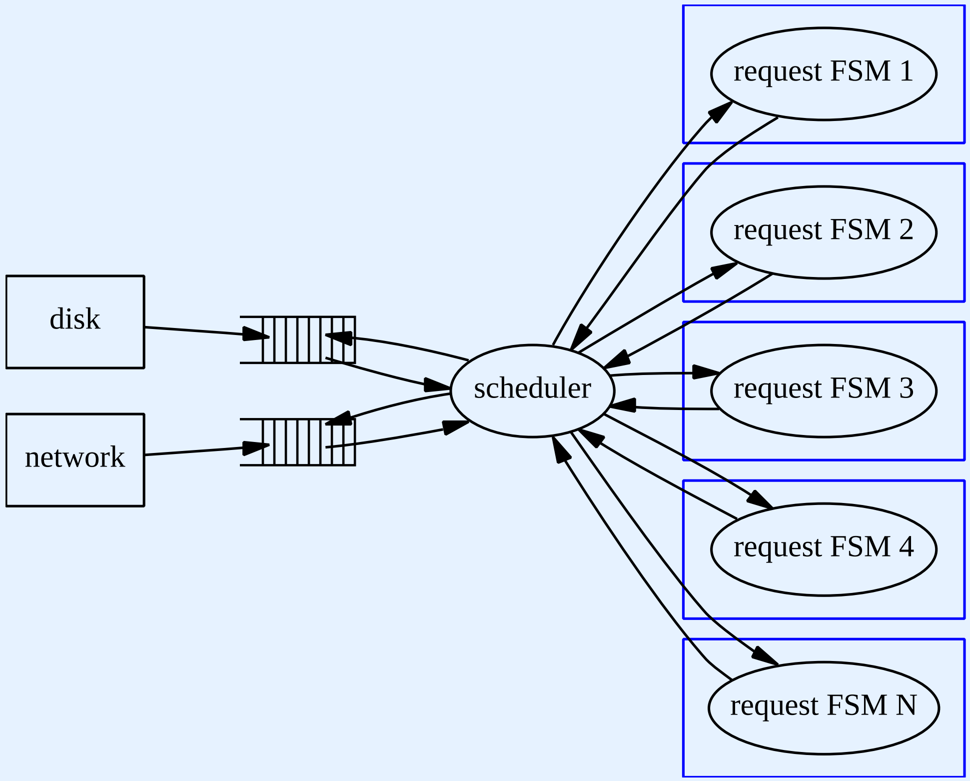
Problems with Threads



(167 MHz uSPARC, Solaris 5.6, 150-byte tasks, $L = 50$ms)

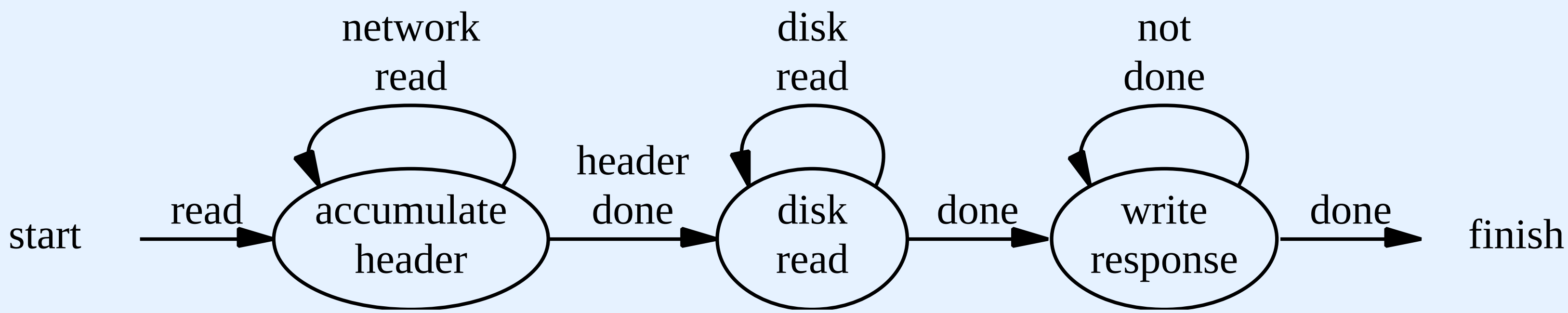
- High resource usage (stacks, etc.)
- High context switch overhead
- Contended locks are expensive
- Too many threads $\longrightarrow$ throughput meltdown

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*

“Monolithic” Event-driven Server



One FSM per HTTP request

- Single thread processes all concurrent requests disjointly

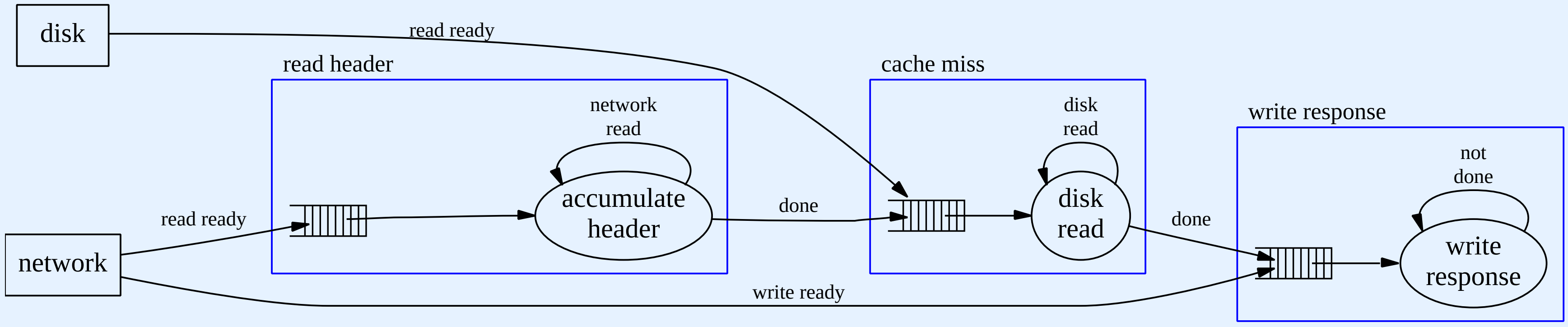
FSM code can never block

- Must use nonblocking I/O
- 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 is some set of states from FSM
- Stages are independent modules
- Queues introduce control boundary for isolation

Threads used to drive stage execution

- Decouples event handling from thread allocation and scheduling
- Stages may block internally
  - *Devote small number of threads to a blocking stage*

SEDA Benefits

Should perform as well as standard event-driven design

- Other optimizations possible:
  - *Delay scheduling of stage until it accumulates work*
  - *Aggregate events to exploit locality*

Support for load conditioning

- Schedule ‘‘high priority’’ stages first during overload
- Can threshold queues to implement backpressure
- Stages can drop, filter, reorder incoming events

Stages can be replicated

- Natural extension to cluster-based design
- Not addressed by this work

Research Issues: Structure and Scheduling

Application structure

- How to decompose an application into stages?
  - *Use a queue or a subroutine call?*
- Queue provides isolation and independent load management
  - *But also increases latency*

Thread allocation and scheduling

- Balance thread allocation across stages based on perceived need
- Tune scheduling algorithms to sustain high throughput
- Interesting algorithms other than priority-based
  - *e.g., wavefront scheduling for cache locality*

Intra-stage event scheduling

- Especially valuable during overload conditions
- Investigate policies such as aggregation and prioritization

Research Issues: Load Conditioning

Least-understood aspect of service design

- Easy: Early rejection of work when overloaded
- Reject at random or according to some policy?
  - ▷ *e.g., allow stock trading orders but not quotes*

Queue thresholding

- How to choose thresholds for queues?
- Interaction with thread scheduler
  - ▷ *refuse to schedule stages upstream from ''clogged'' stage*

Resource management

- Imagine fast stage which allocates a lot of memory
- Need to perform per-stage resource management
  - ▷ *cf. Resource containers, Scout OS*

Research Issues: Debugging

Difficulty in event-driven systems

- Thread stack no longer represents individual task processing
- Existing debugging tools assume thread-based model

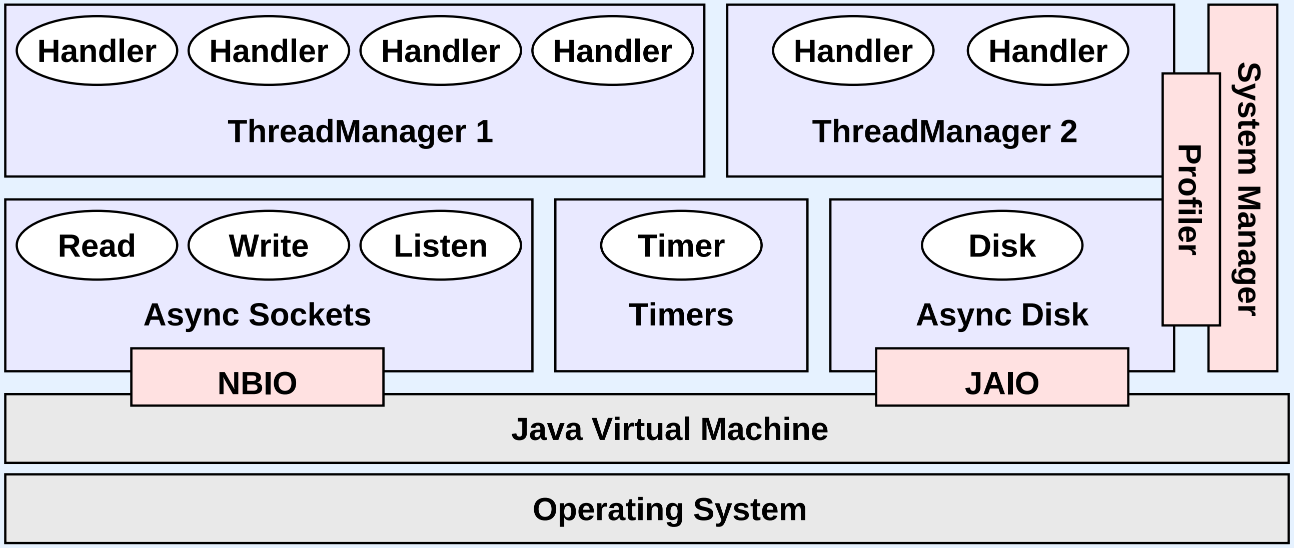
SEDA design can help

- Tools to visualize queue lengths over time
- Tools to visualize stage connectivity and event flow

Sandstorm prototype has both

- Rudimentary but very useful

SEDA Prototype: Sandstorm



Event handlers

- Core application logic for stages
- Simple interface: `handleEvents()`, `init()`, `destroy()`

Implemented in Java with nonblocking I/O interfaces

- NBIO: Nonblocking socket I/O and `select()`
- JAIO: Nonblocking disk I/O via POSIX.4 AIO

Thread Manager Interface

Key aspect of Sandstorm design

- Allocate and schedule threads to drive stage execution
- Decouples threading policy from application code

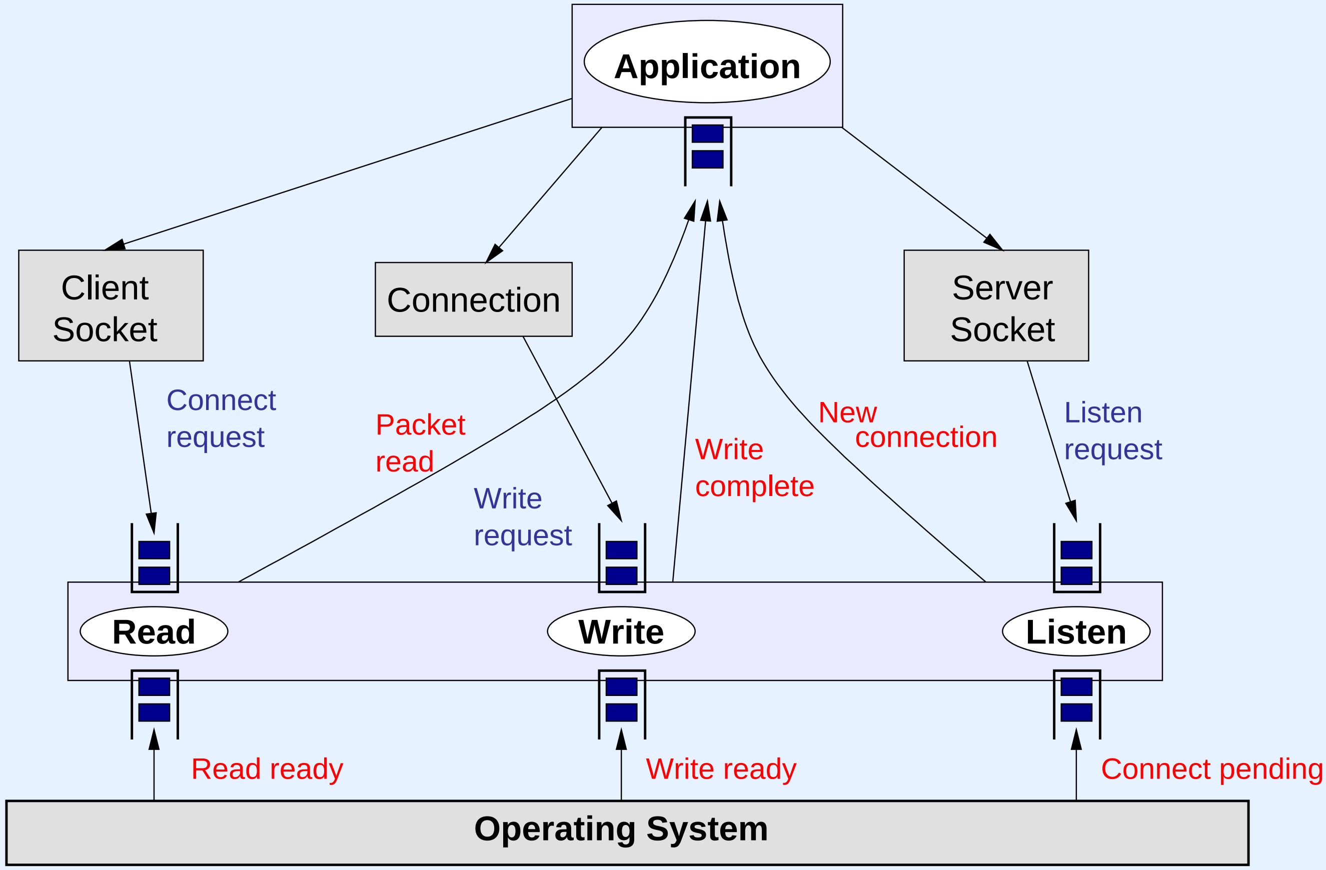
Thread per processor (TPP) implementation

- One thread per physical CPU
- Threads process stages in round-robin fashion
- Many extensions possible:
    - ▷ *e.g. Schedule stages along event dispatch path*

Thread per stage (TPS) implementation

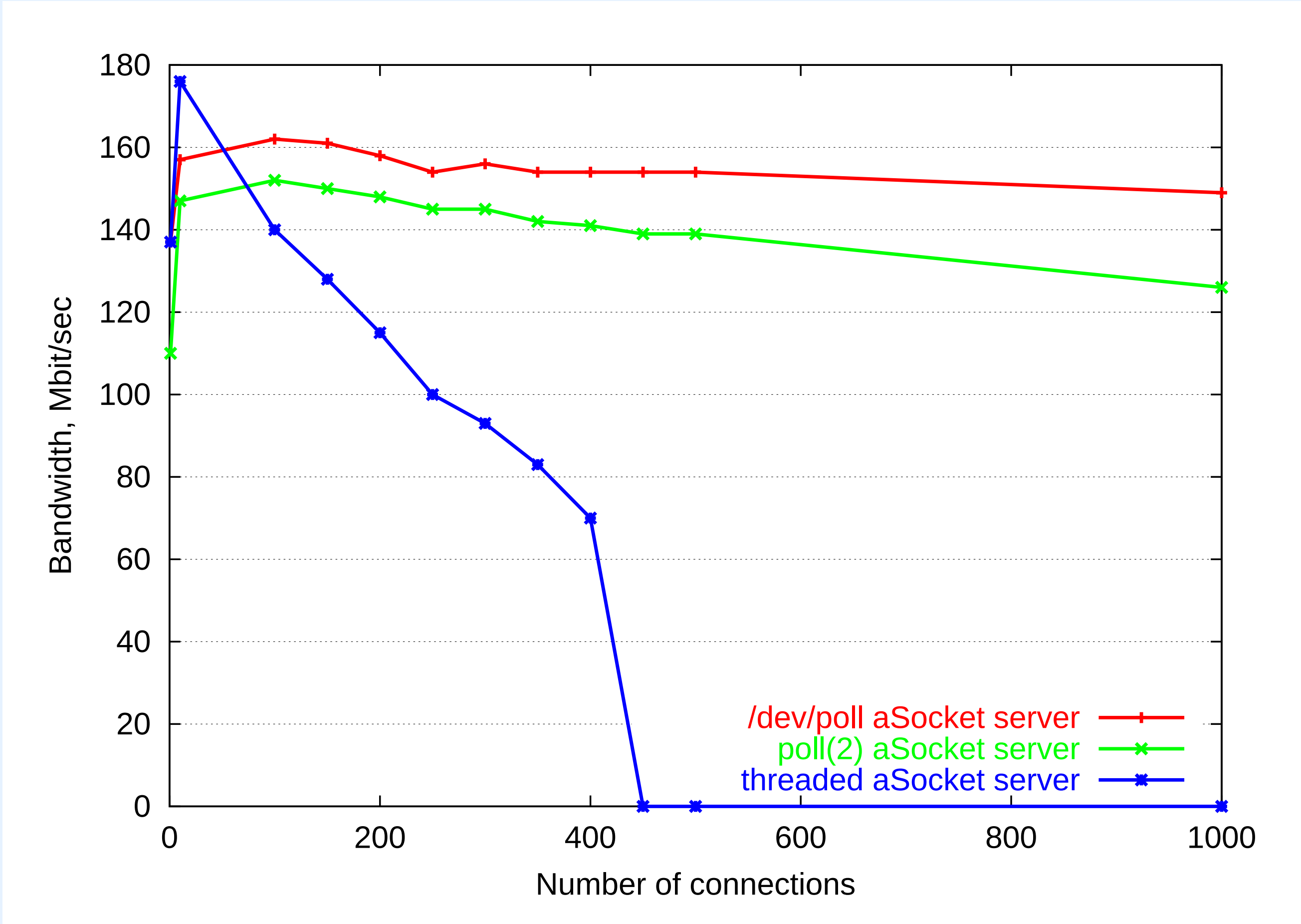
- One thread per event queue per stage
- Each thread blocks on its queue
- Relies on O/S level scheduling for stage prioritization

Asynchronous Sockets Layer



- Three stages: read, write, listen
- Controlled by own thread manager
- Application enqueues connect, write, and listen events
- Sockets layer pushes up packets and connections

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
- Sandstorm obtains *100 Mbps* for *10,000* connections

Other Sandstorm Components

Asynchronous Disk Layer

- Still under development with James Hendricks
- Based on Java interface to POSIX.4 AIO calls
- Efficient (we hope) Linux implementation available
- Design analogous to asynchronous sockets

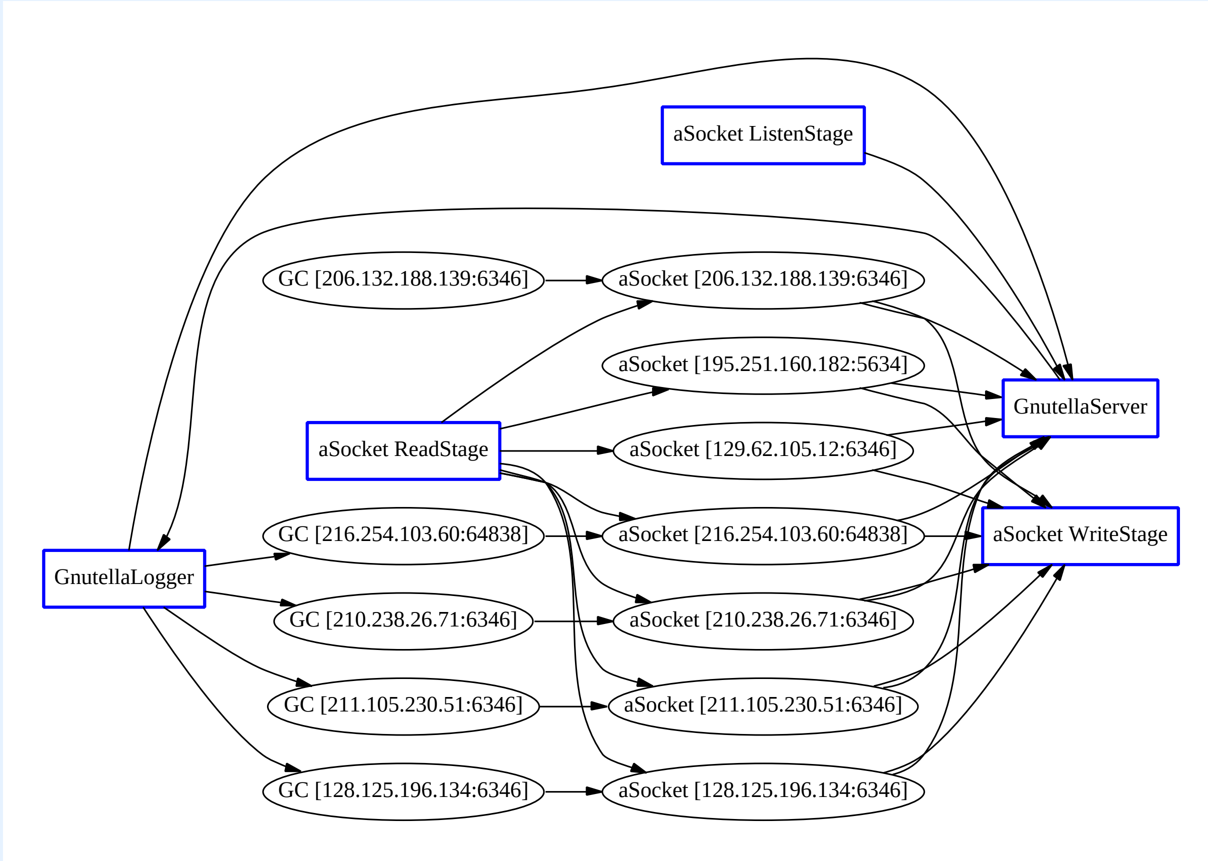
Timers

- Stages register events to fire at some later time
- Implemented as stage with own thread

System Manager Interface

- Used by stages to obtain handle to other event queues
- Also used to dynamically create and destroy stages

Sandstorm Profiler



Automatic visualization of stage connectivity

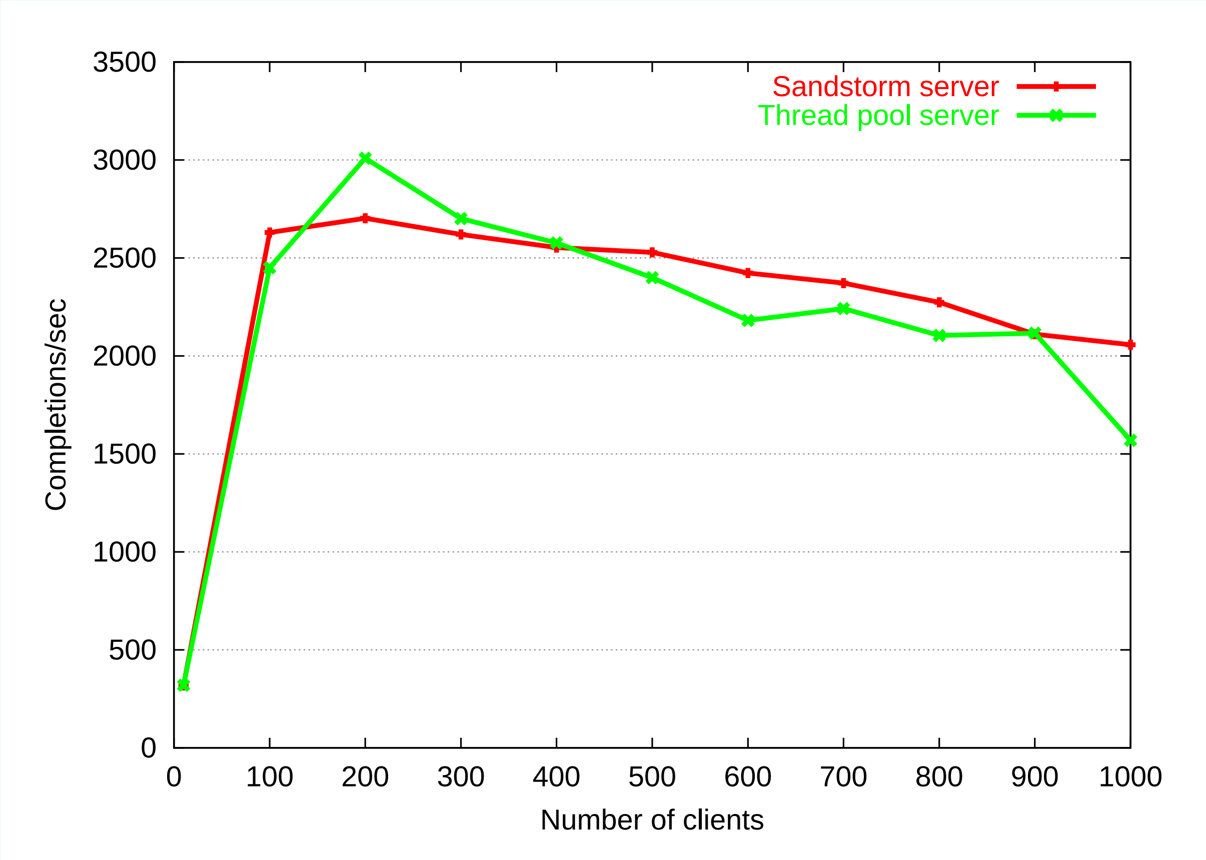
- Nodes represent stages or event-processing classes
- Edges represent event dispatch paths

Temporal trace of queue lengths (more later)

Outline of this Talk

- Existing Concurrency Models
- SEDA Architecture Overview
- Research Issues
- Initial prototype: Sandstorm
- Application Evaluation: HTTP and Gnutella
- Related Work and Timeline

HTTP Server Benchmark



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

- Return 8Kb webpage from memory, clients sleep 20 ms
- Sandstorm server uses single stage
- Threadpool server uses 150 threads
  - ▷ *HTTP/1.1 Persistent Connections, 100 requests/conn*

What's Wrong With This Picture

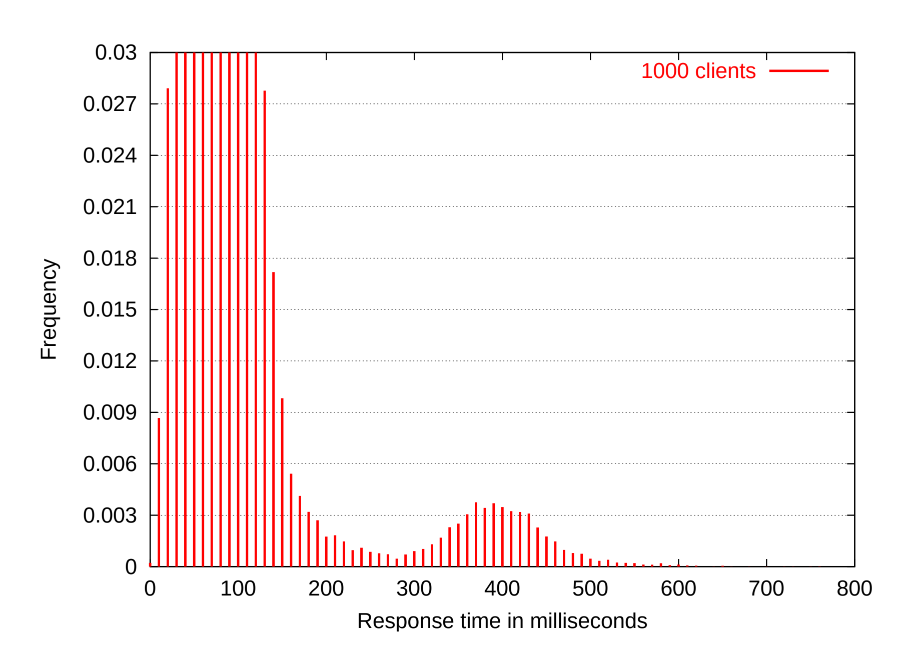
It ignores response time!

- SEDA and threaded servers both sustain high throughput
- But threaded server has limited capacity: 150 threads

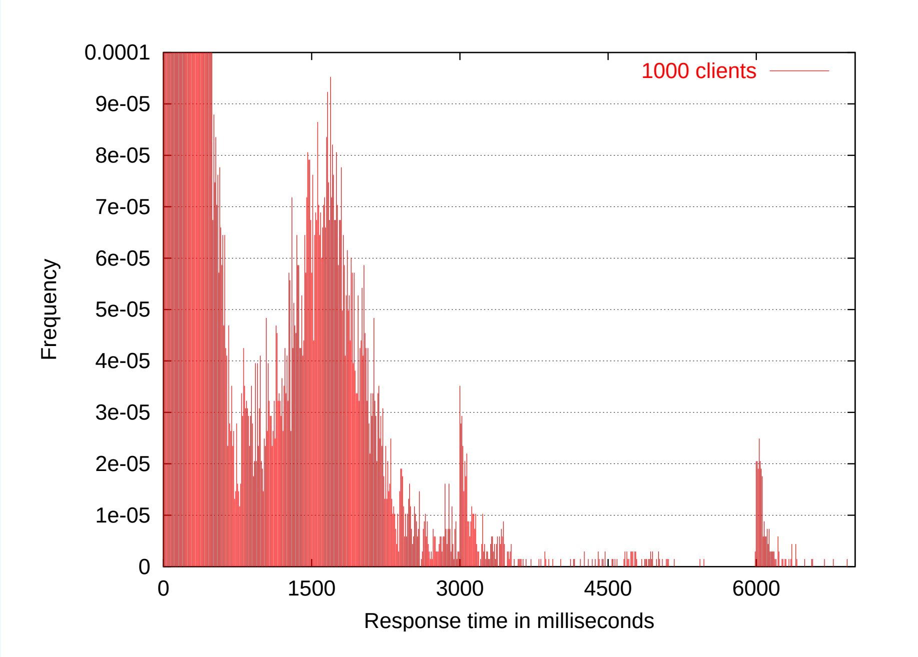
Must use alternate metrics

- ''Latency does matter''
- Response time, connect time
- Fraction of clients serviced per unit time
- SPECweb99: Number of simultaneous conns obtaining certain bandwidth

Response Time Histograms



Sandstorm



Threadpool

- Sandstorm: median *1105 ms*, max *15344 ms*
- Threaded: median *4570 ms*, max *190766 ms*

- To be done: Build real Web server, use industry-standard benchmark

Gnutella Packet Router

Goal: Explore application domains other than client/server

- Different properties and challenges

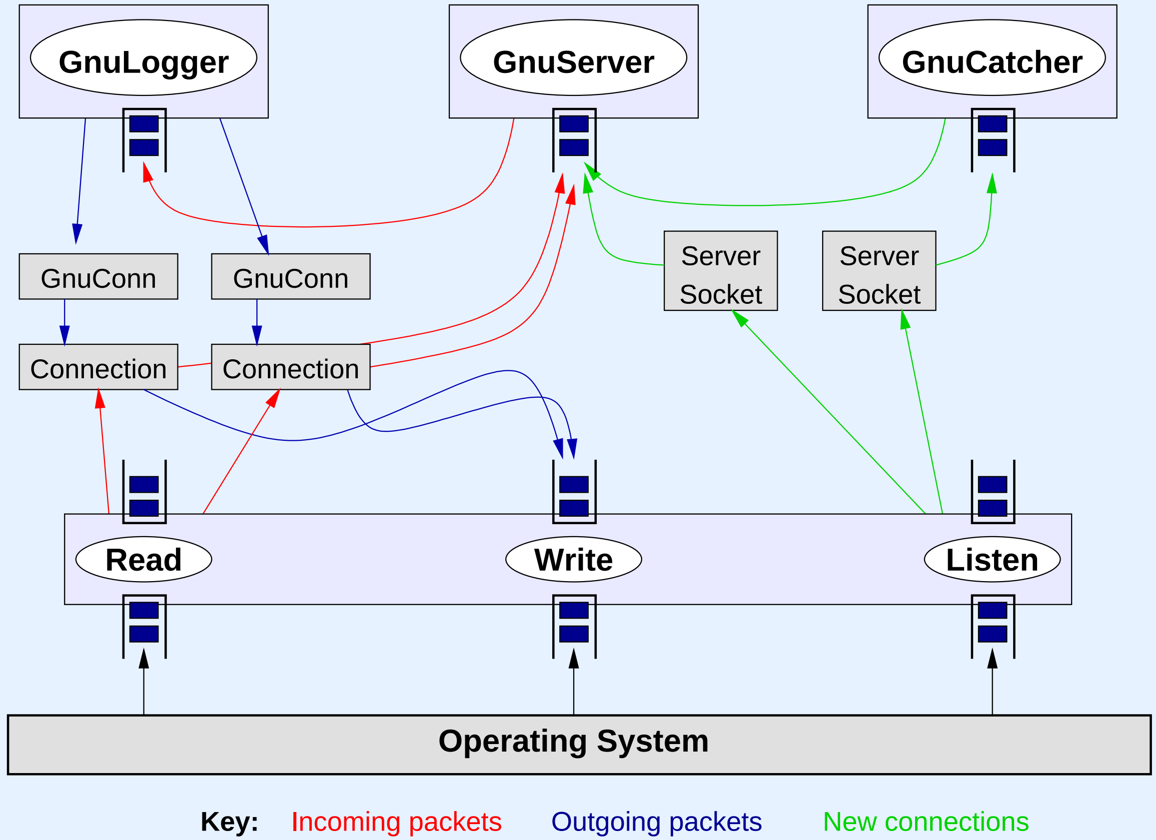
Goal: Demonstrate load conditioning

- Introduce bottleneck into server
- Exhibit good behavior under heavy load

Gnutella basics

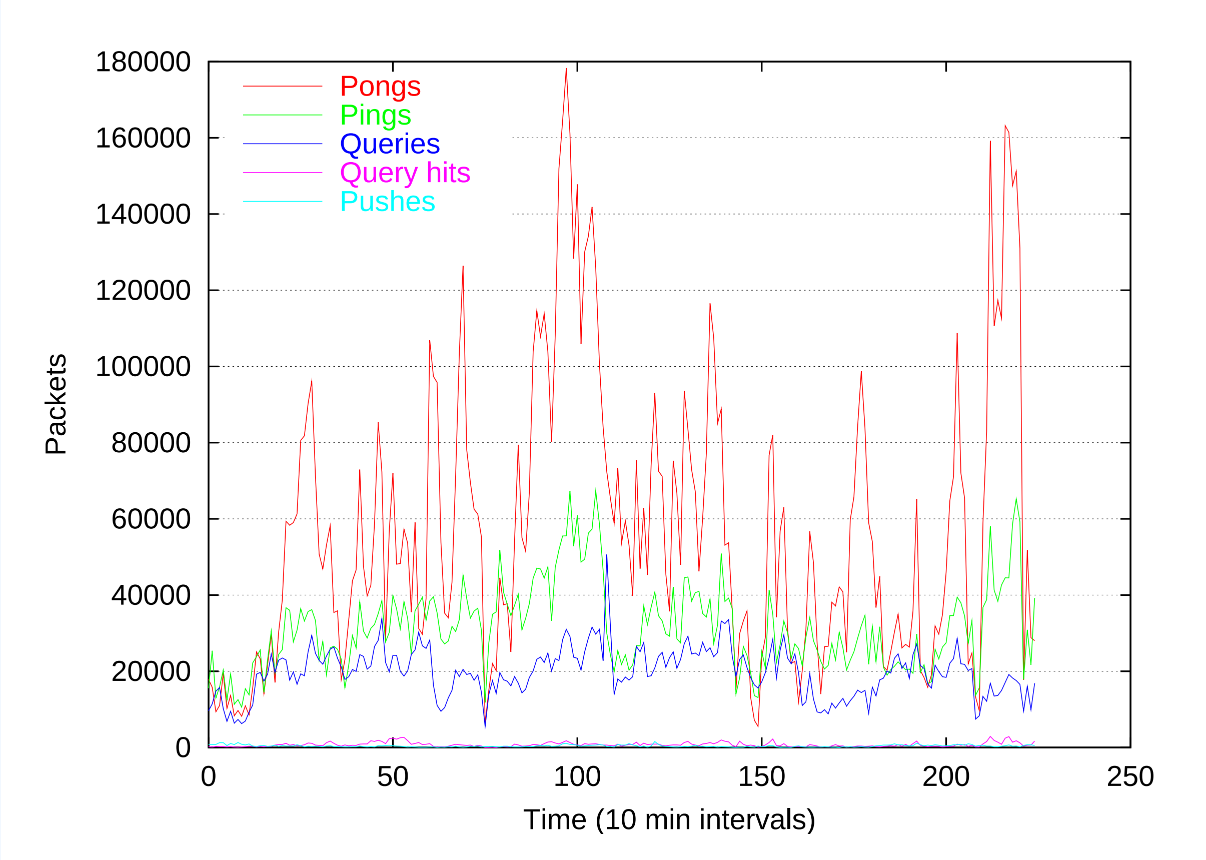
- Decentralized peer-to-peer file sharing network
- Every node exchanges messages with its neighbors
  - ▷ *ping, pong, query, queryhit, push message types*
- Direct download from host via HTTP
- Initial discovery via well-known host
- Several thousand users at any time, 10’s of TBs of data

Sandstorm Gnutella Server



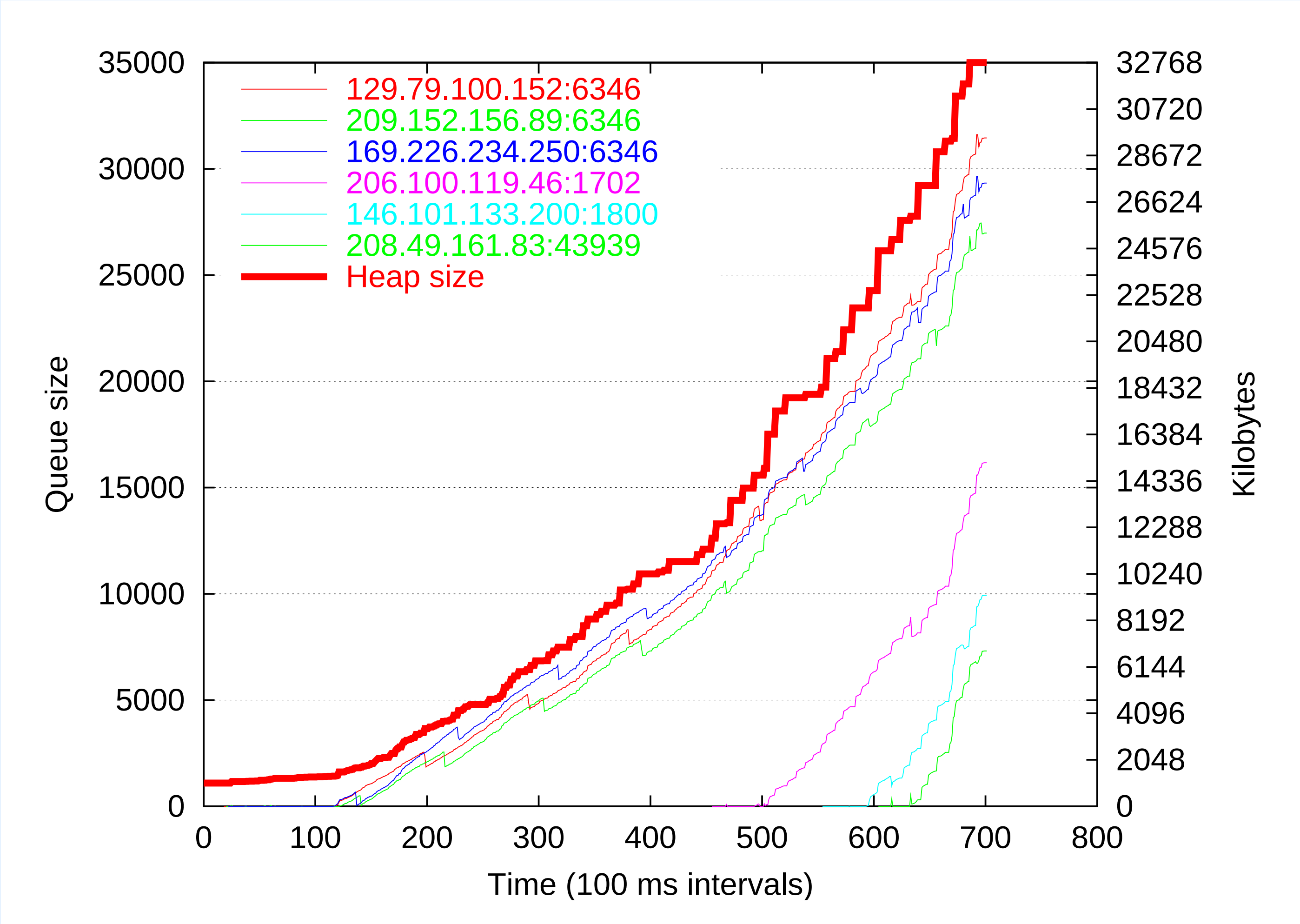
- Logger: Routes and logs packets
- Server: Parses incoming packets
- Catcher: Establishes new connections
- Gnutella Connection: Formats outgoing packets

Gnutella Packet Trace



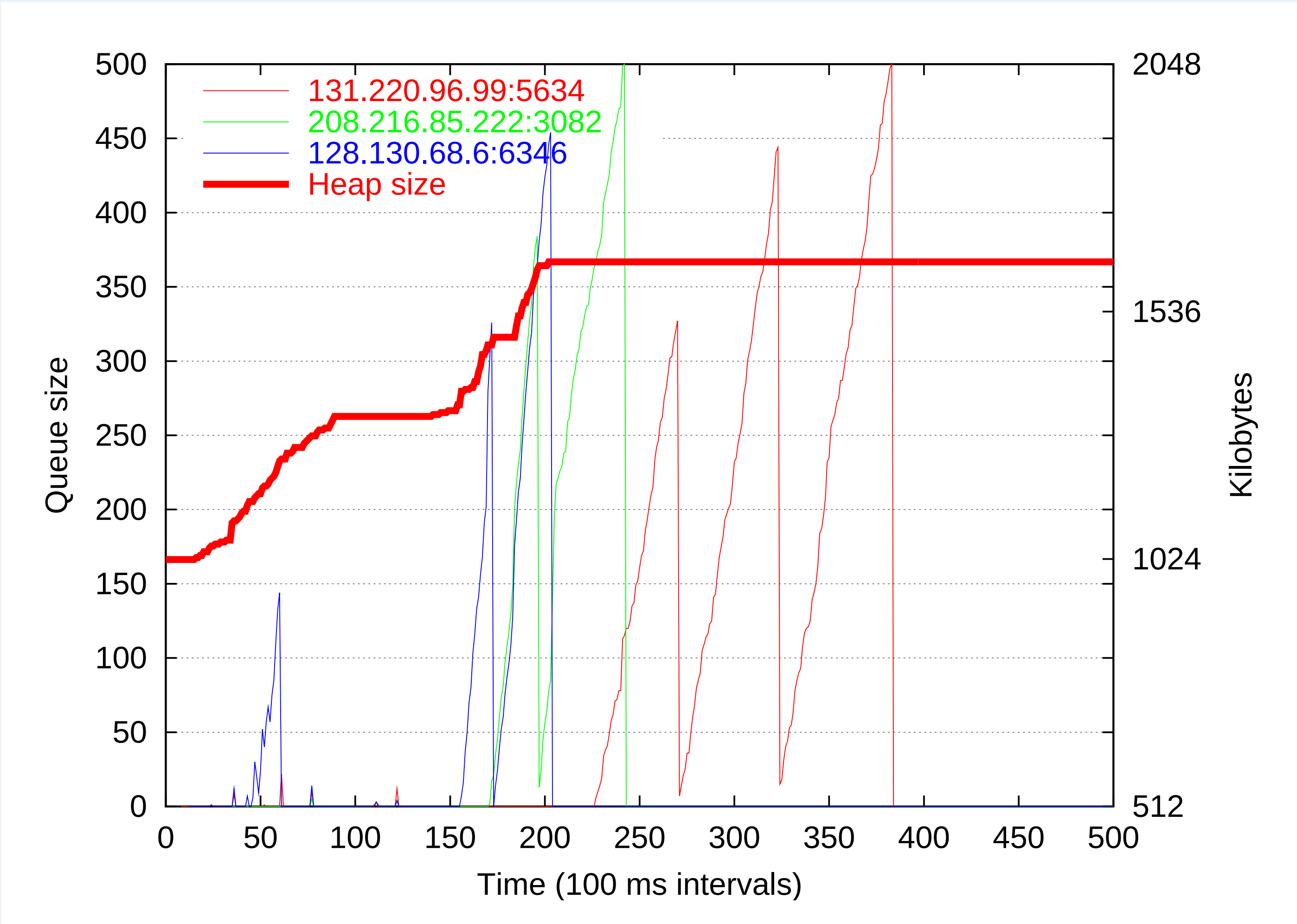
- Gathered over 37-hour period
- *24.8 million* packets, average *179.55* per sec
- *72396* connections, average 12 at any time
- Very bursty, no clear diurnal pattern

Dealing with Clogged Connections



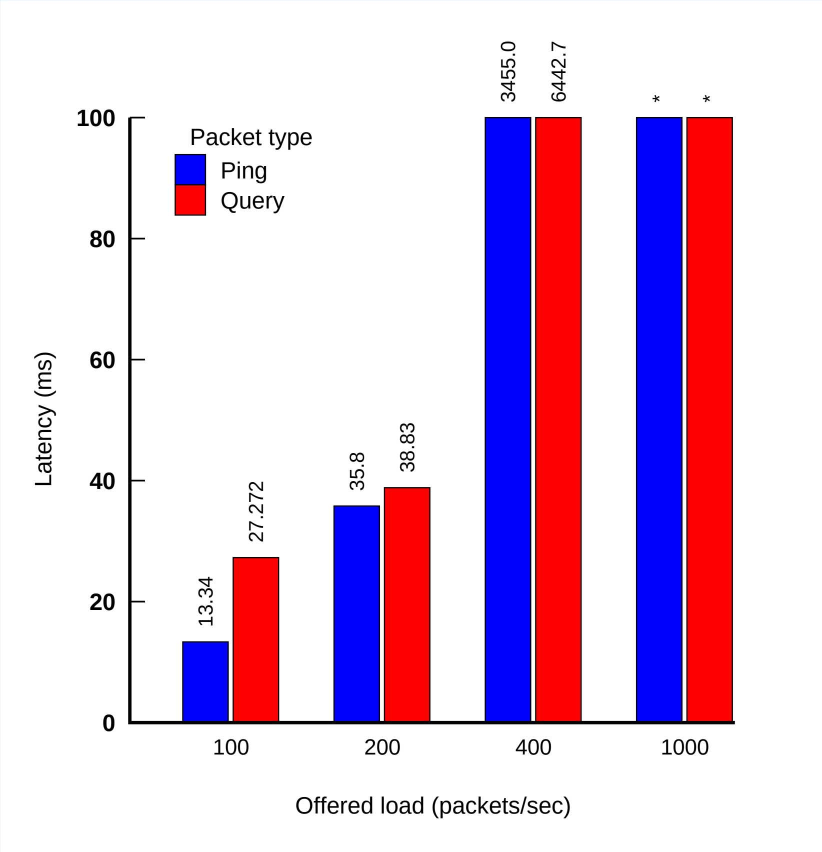
- Server would crash after a few hours
- Cause: saturated connections
  - ▷ *115 packets/sec can saturate a 28.8 modem link*

Socket Queue Thresholding



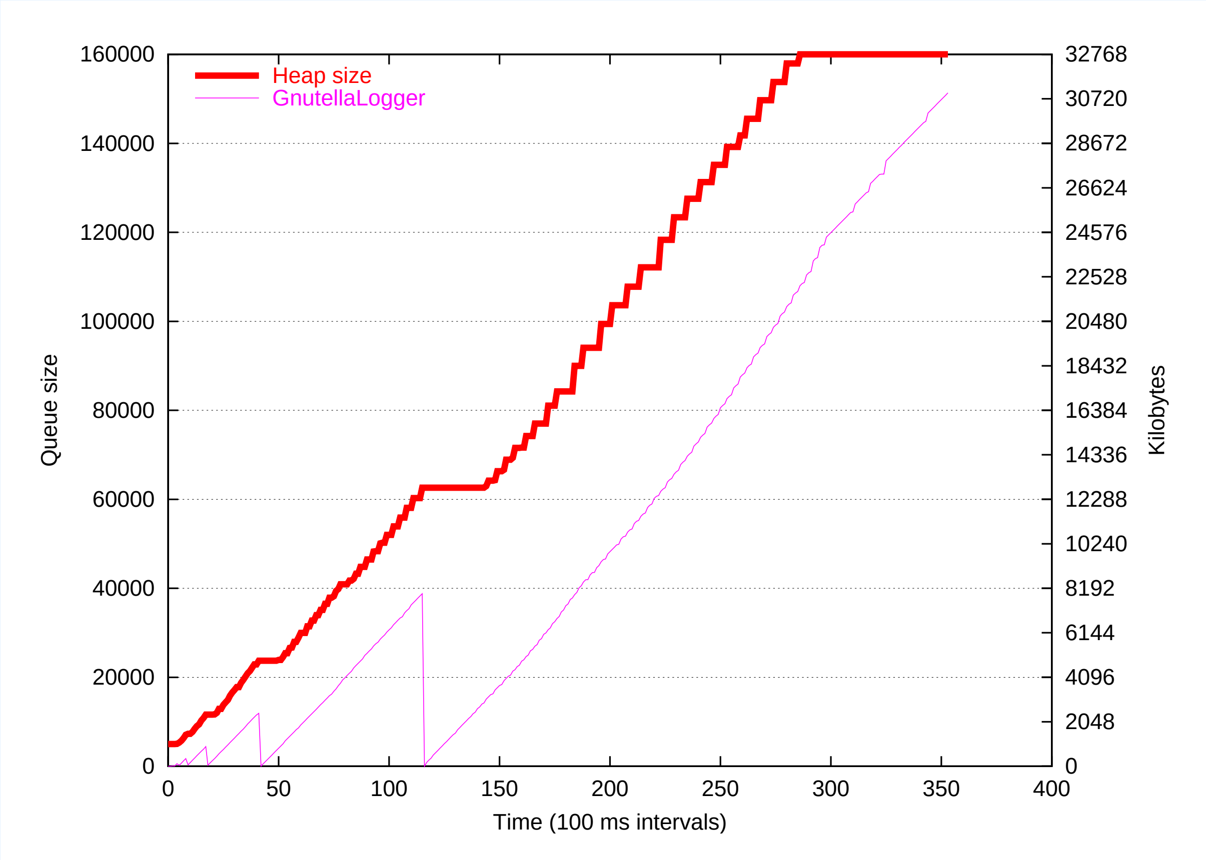
- Close connection if outgoing queue reaches threshold
- One form of load conditioning
- Many variations possible

Router Latency Under Overload Condition



- Benchmark client generates realistic packet streams
- Introduce intentional bottleneck into server:
  - ▷ *Server-side delay of 20 ms for query messages*
  - ▷ *15% of messages are queries*
- Server crashed at 1000 packets/sec

Sandstorm Profile of Overloaded Server



- Offered load of 1000 packets/sec
  - *Query message delay of 20 ms*
- Clearly indicates *GnutellaLogger* stage as bottleneck

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

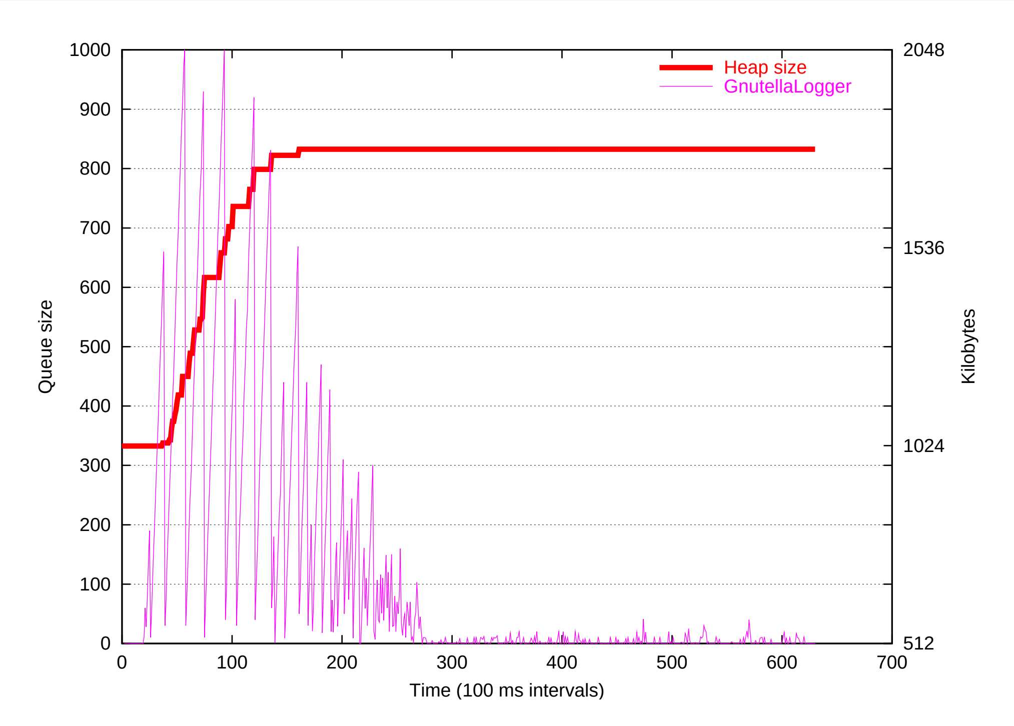
Thread pool resizing

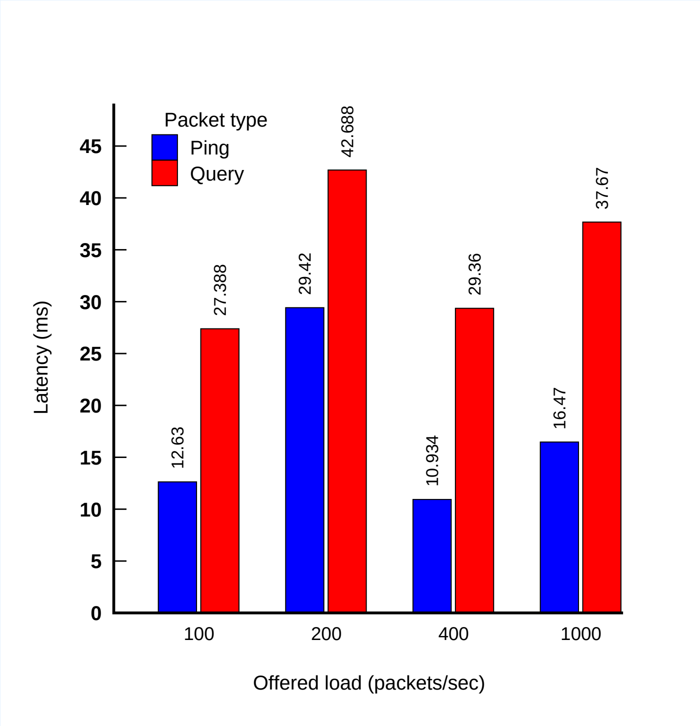
- Devote more threads to bottleneck stage
- Model stage as G/G/n queueing system
- n threads, arrival rate λ, query frequency p, query servicing delay L

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

▷ *Unfortunately, can't determine a priori*

Sandstorm Thread Governor





- 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

- *[Flash, Harvest, Squid, JAWS, ...]*
- Mostly ‘‘monolithic’’ event-driven systems; some SEDA-like
- Little work on load conditioning, event scheduling

StagedServer (Microsoft Research)

- Uses SEDA-based design
- Primarily concerned with cache locality
- Simple wavefront thread scheduler only

Click Modular Router, Scout OS, Utah Janos

- 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
- Can apply this approach to SEDA

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

Research Methodology

Performance and load analysis of applications

- Traditional apps: Web servers, SPECweb99, TPC-W
- Nontraditional apps: Gnutella, Music Search Engine
- Evaluate performance, load conditioning, ease of programming
- Contrast to standard threaded and event-driven models

Incorporation into Ninja and OceanStore

- Sandstorm as basis of Ninja clustered services platform
- Hopeful adoption as base for OceanStore storage manager
- Telegraph?

Release to world, measure impact

- Full release of all software in 6-12 months
- NBIO and other components already available
- Influence on Sun JSR for new I/O APIs in Java

Timeline

0-6 months

- Continue development of Sandstorm prototype
- Investigate scheduling and load conditioning policies
- Complete asynchronous disk layer
- Develop dynamic HTTP server
- Submit to SOSP

6-12 months

- Develop second application: Gnutella-based music search engine
- Use app to drive Sandstorm prototype
- Work with Ninja and OceanStore to encourage adoption
- Initial public release

12-18 months

- Incorporate feedback into next revision
- Develop debugging and visualization tools
- Write thesis and graduate

Summary

Staged Event-Driven Architecture designed to support

- High concurrency
- Good behavior under heavy load
- Modularity and code reuse

Lots of interesting research directions

- Application structure
- Thread and event scheduling
- Load conditioning policies
- Programming and debugging tools

Promising initial results

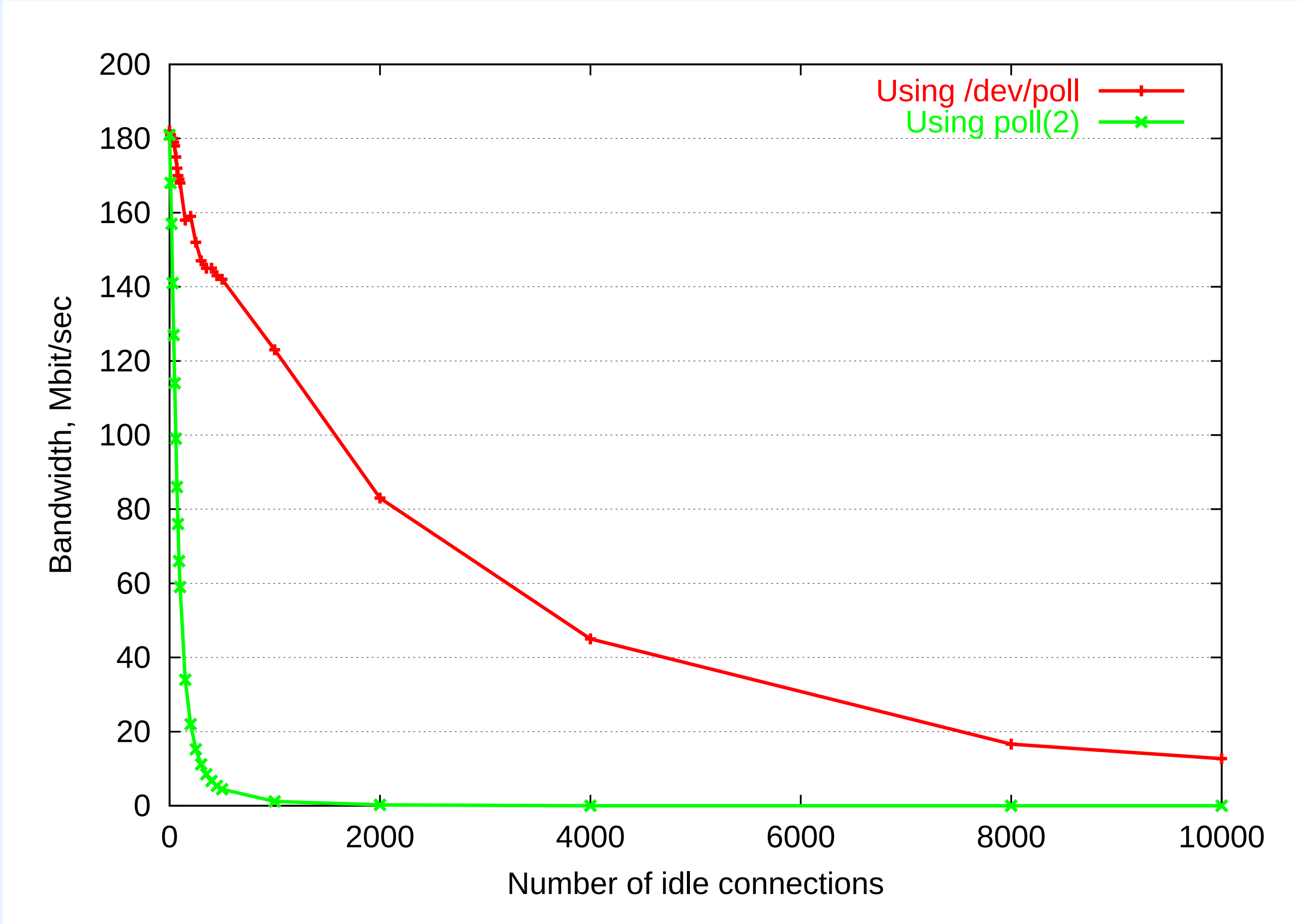
- *Sandstorm* service platform
- Application scalability and load conditioning

For more information

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

Backup Slides Follow

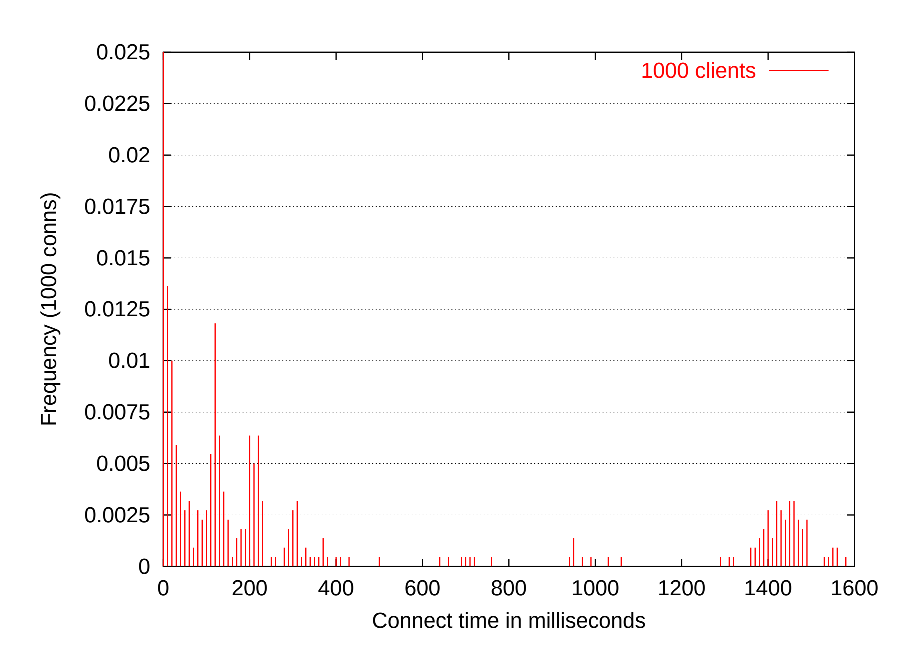
Effect of Idle Connections



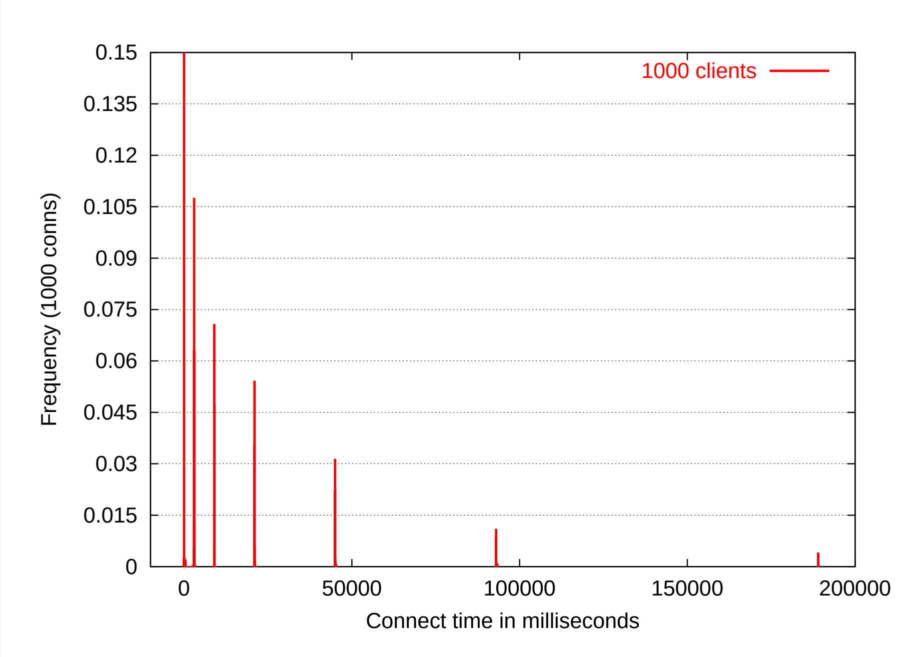
(4-way 500 MHz PIII, Gigabit Ethernet, Linux, IBM JDK 1.1.8)

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

Connect Time Histograms



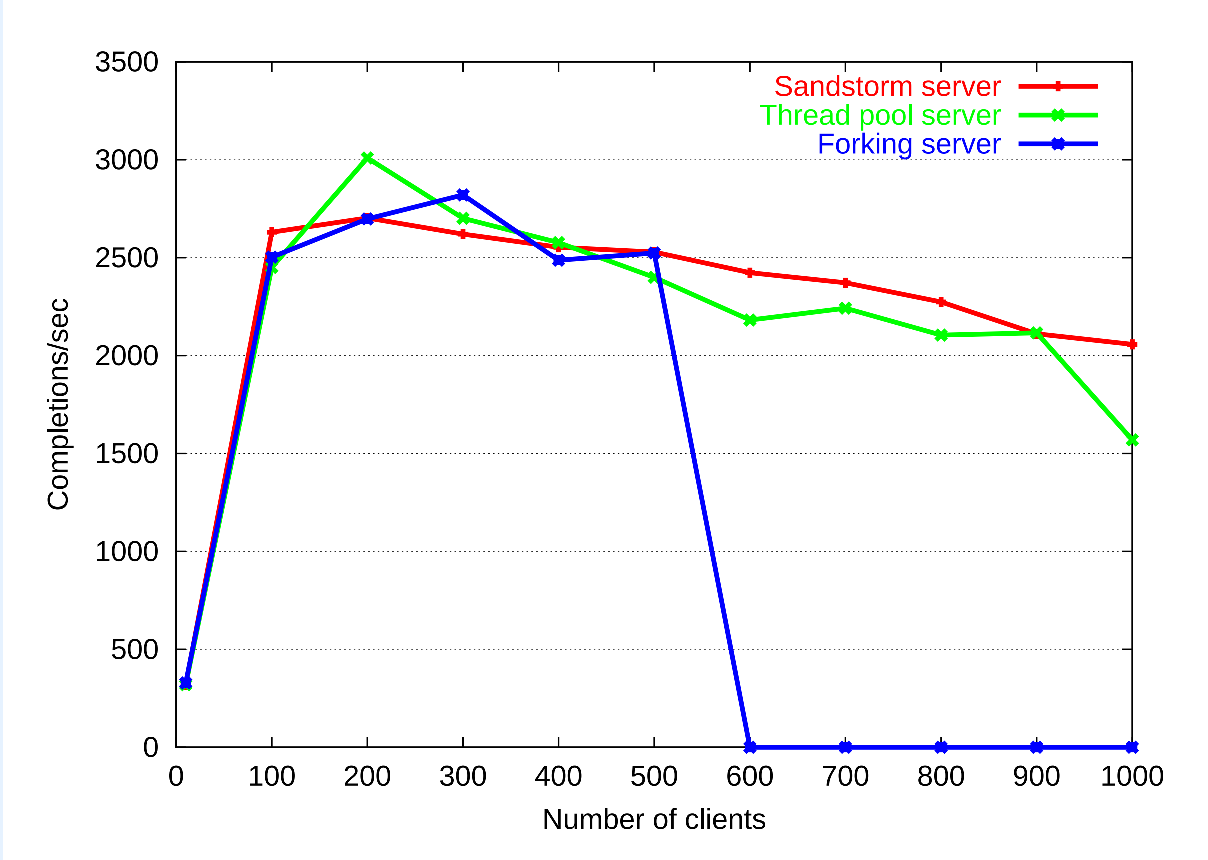
Sandstorm



Threadpool

- Sandstorm: median *420 ms*, max *3116 ms*
- Threaded: median *3105 ms*, max *189340 ms*

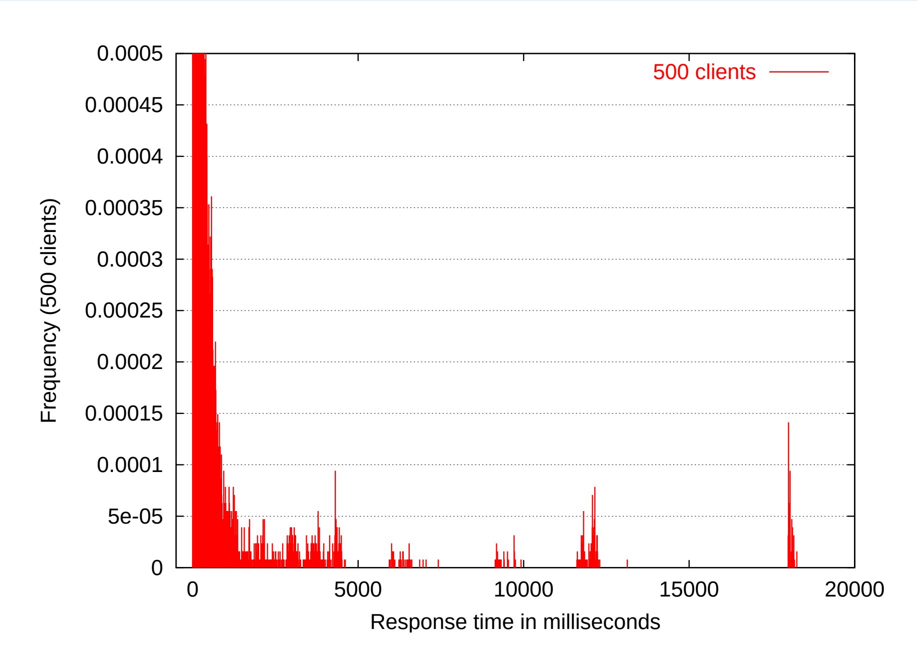
Forking HTTP Server Throughput



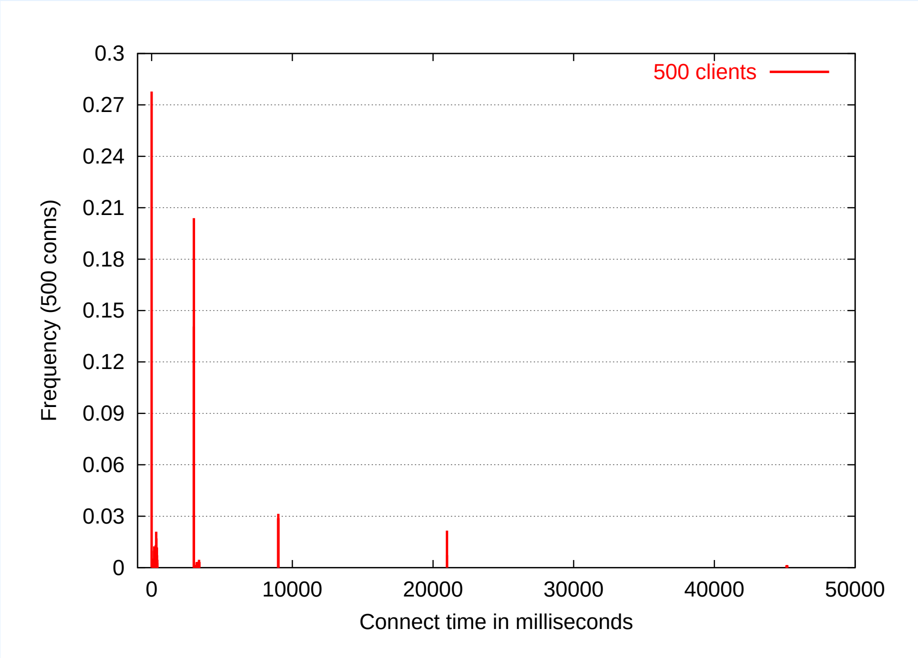
(4-way 500 MHz PIII, Gigabit Ethernet, Linux, IBM JDK 1.1.8)

- Sandstorm: 3 threads, nonblocking I/O
- Threadpool: 150 threads, blocking I/O
- Forking: one thread per connection, blocking I/O

Forking HTTP Server



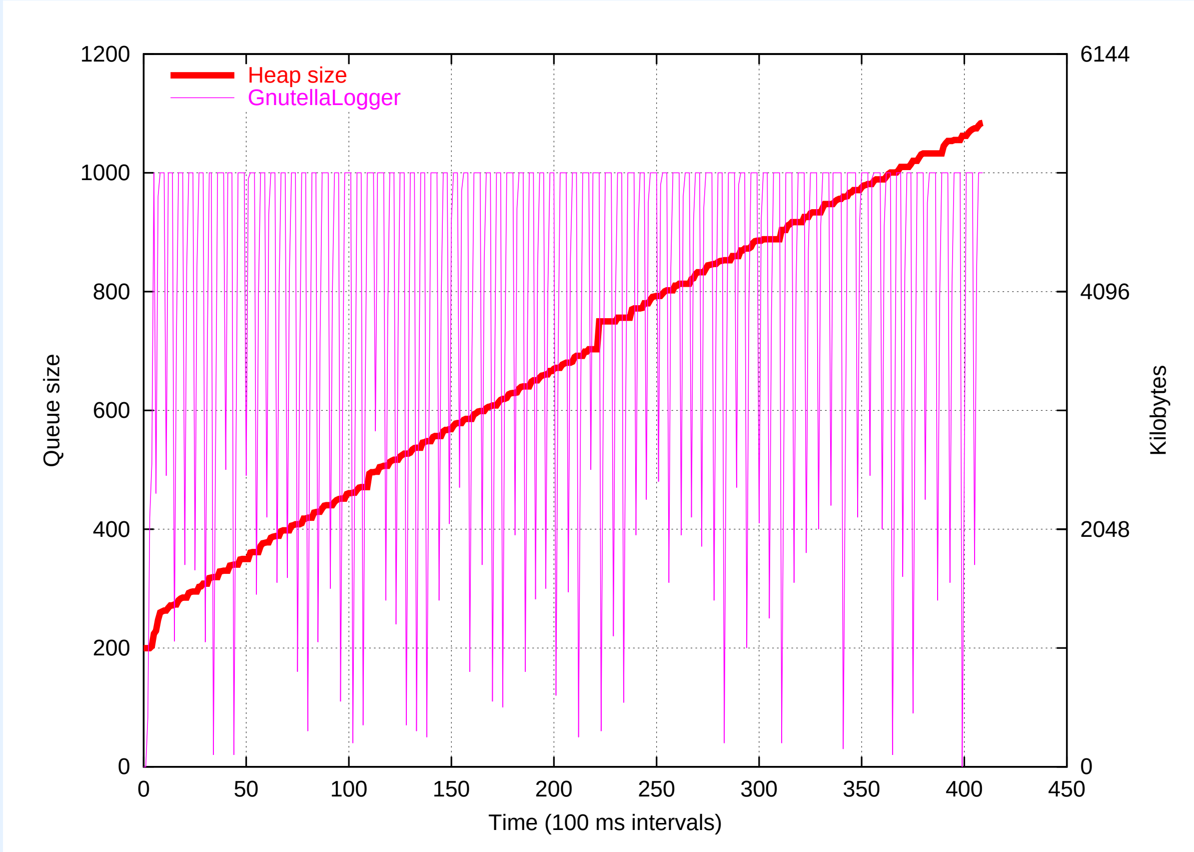
Response time



Connect time

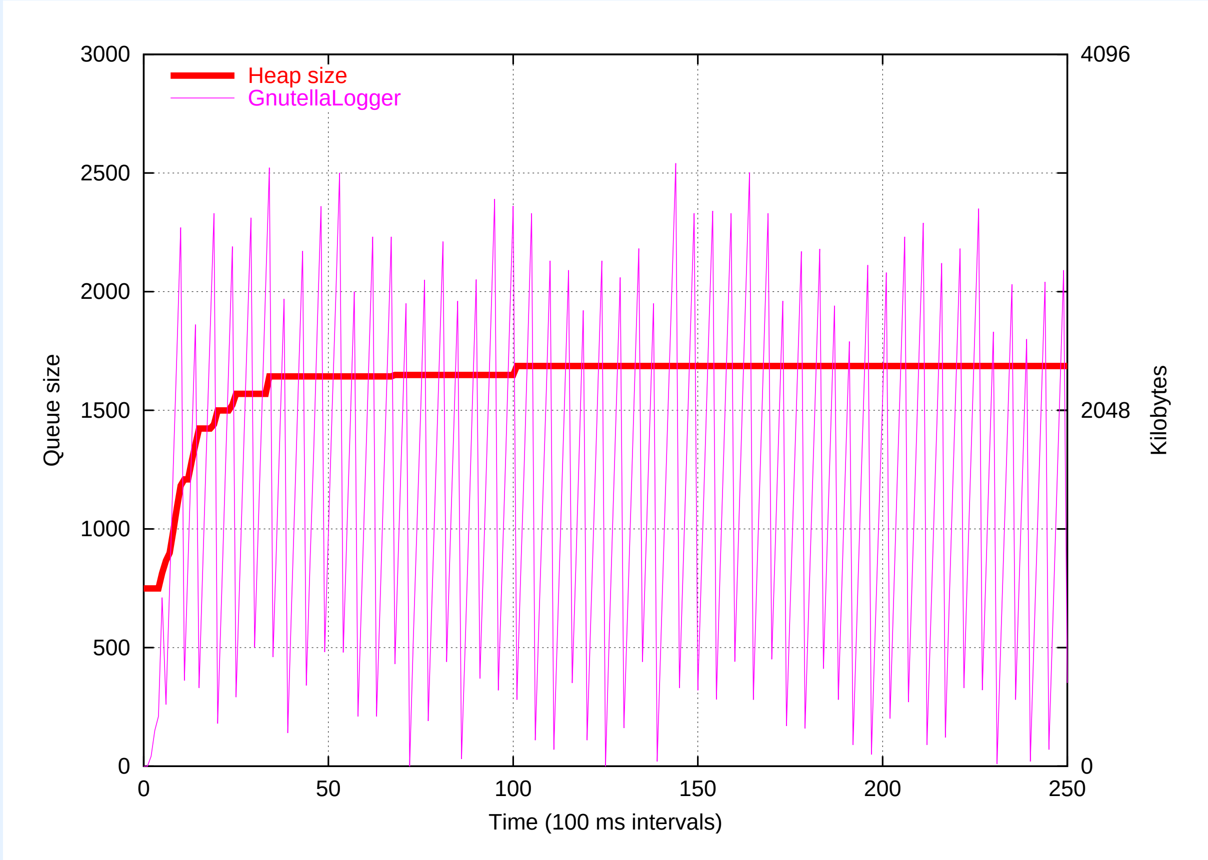
- Response time: median *2920 ms*, max *48136 ms*
- Connect time: median *2990 ms*, max *45201 ms*

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