Performance Aspects of the Staged Event-Driven Architecture

Matt Welsh

mdw@cs.berkeley.edu

Ninja Retreat, Lake Tahoe
January 10, 2001

Highly Concurrent Server Challenges

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

Proposal for a New Architecture

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

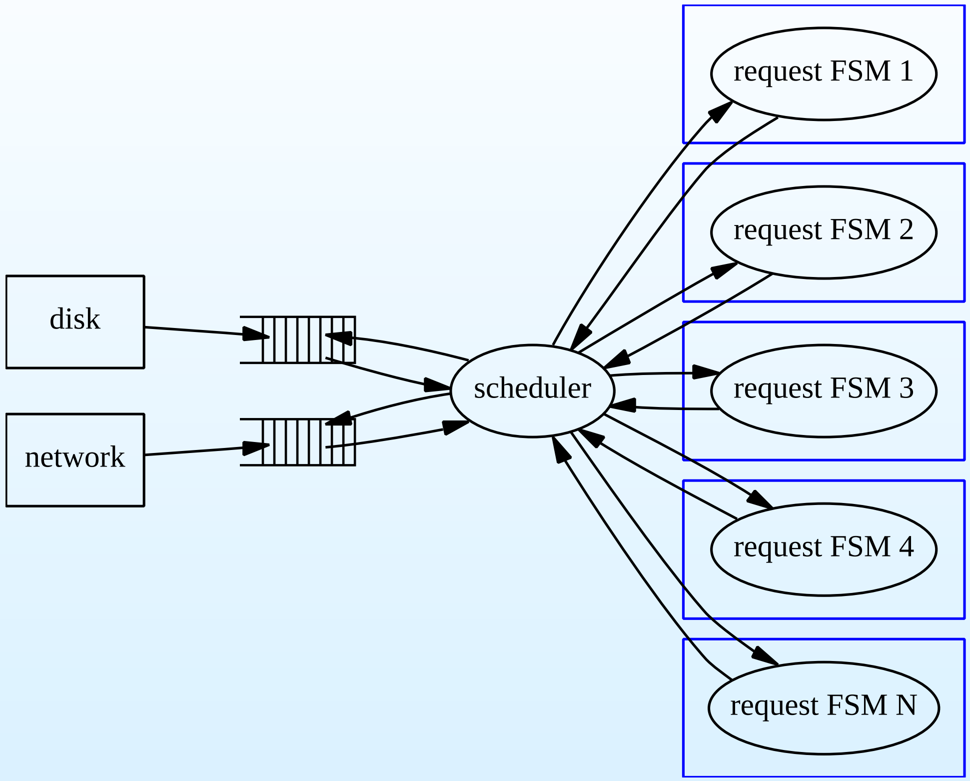
Enable load conditioning

- 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

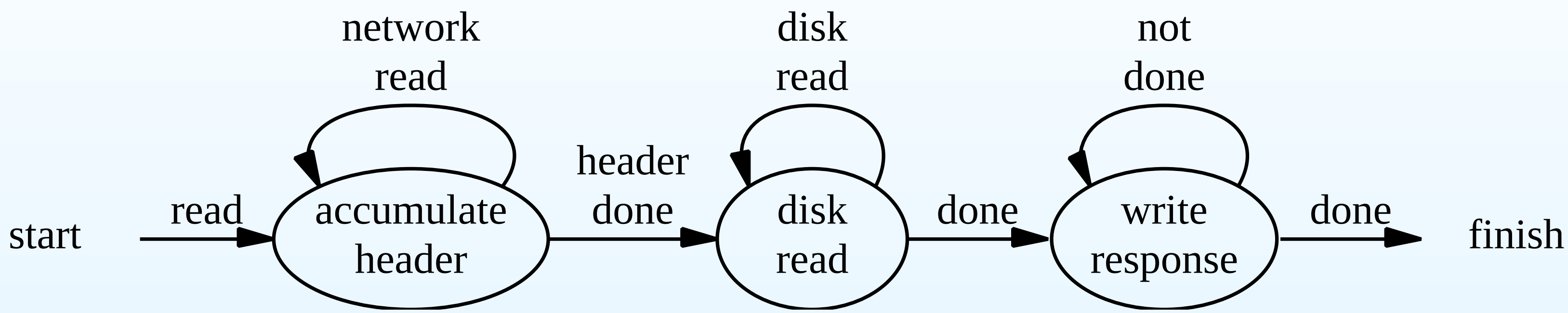
- Not just tuned for specific app (e.g., Web servers)
- General-purpose architecture for servers

"Monolithic" 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*

Simple Event-driven HTTP Server



One FSM per HTTP request

- Single thread processes all concurrent requests disjointly

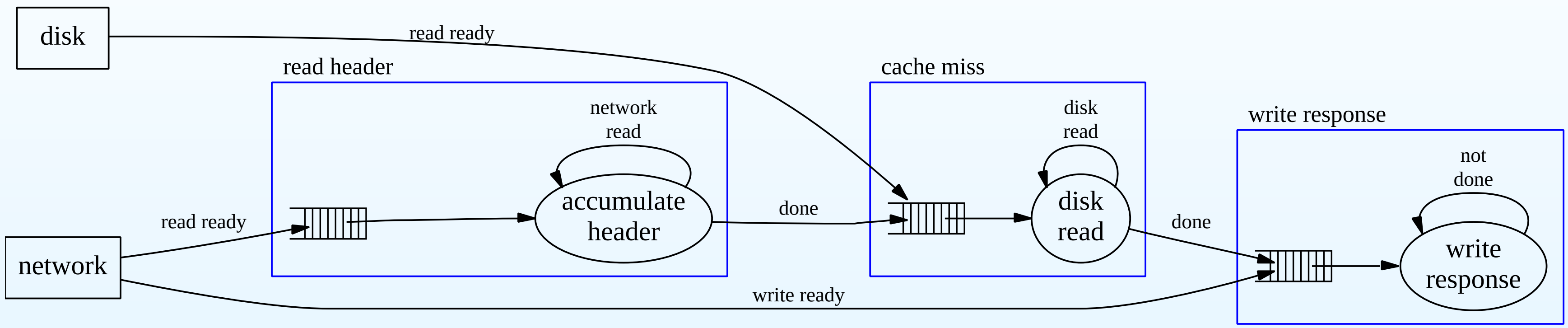
FSM code can never block

- 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*

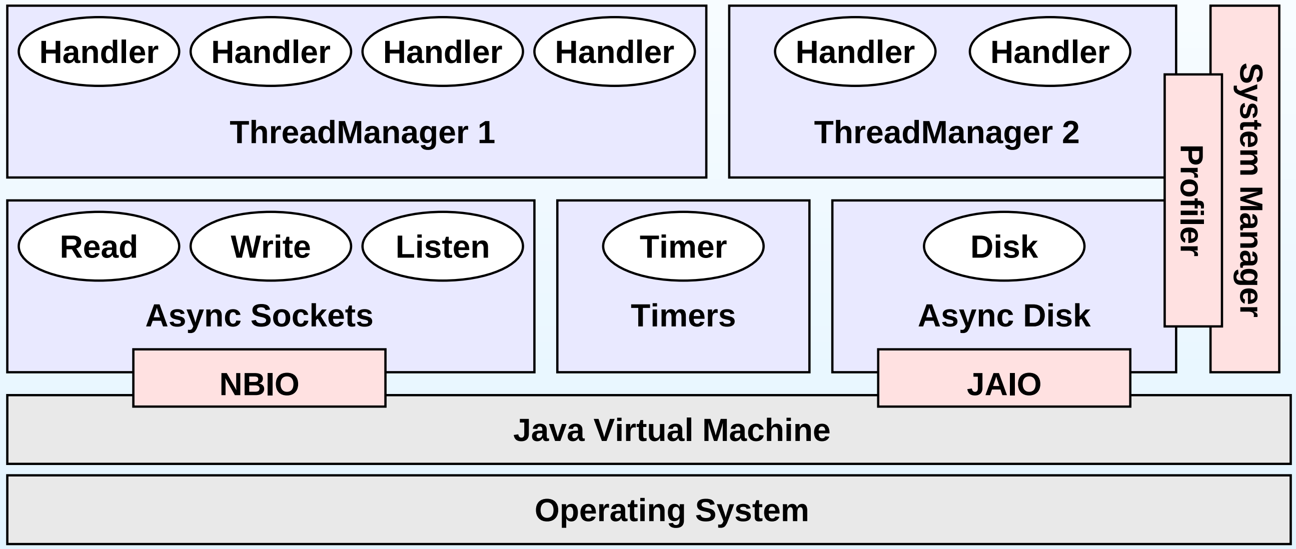
Queue thresholding

- 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*

SEDA Prototype: Sandstorm



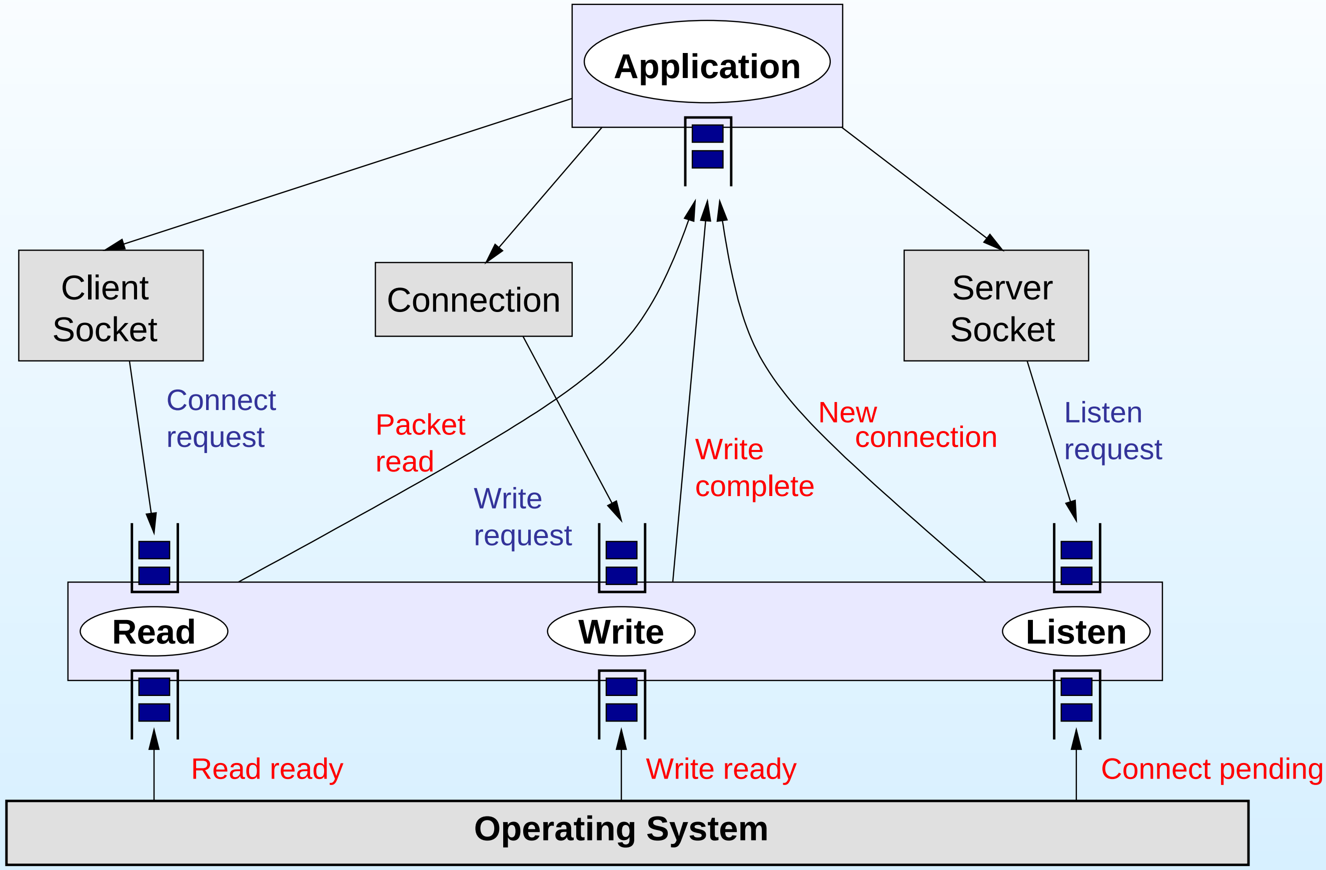
Event handlers

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

Implemented in Java with nonblocking I/O interfaces

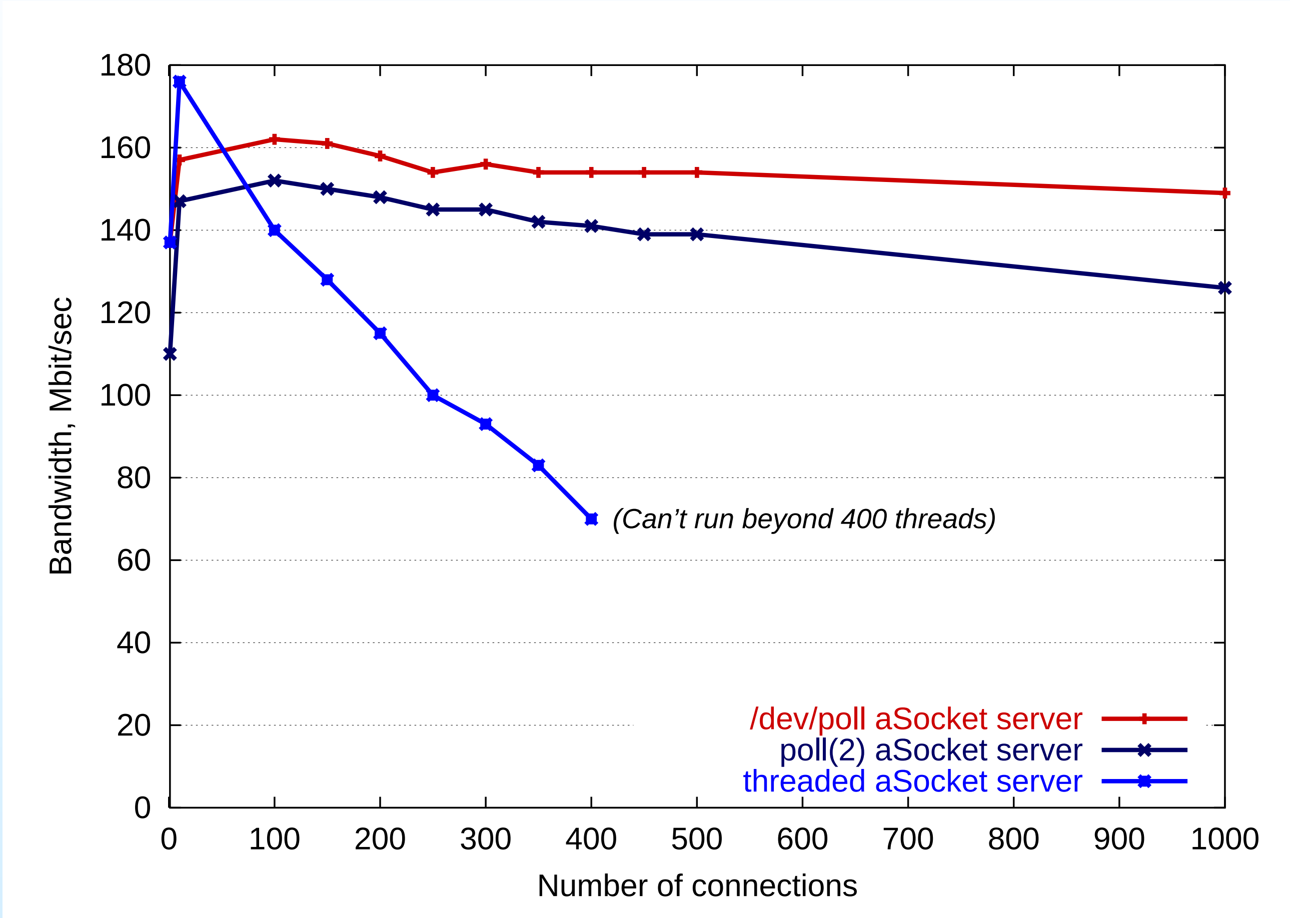
- NBIO: Nonblocking socket I/O and `select()`
- JAIO: Nonblocking disk I/O via POSIX.4 AIO

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

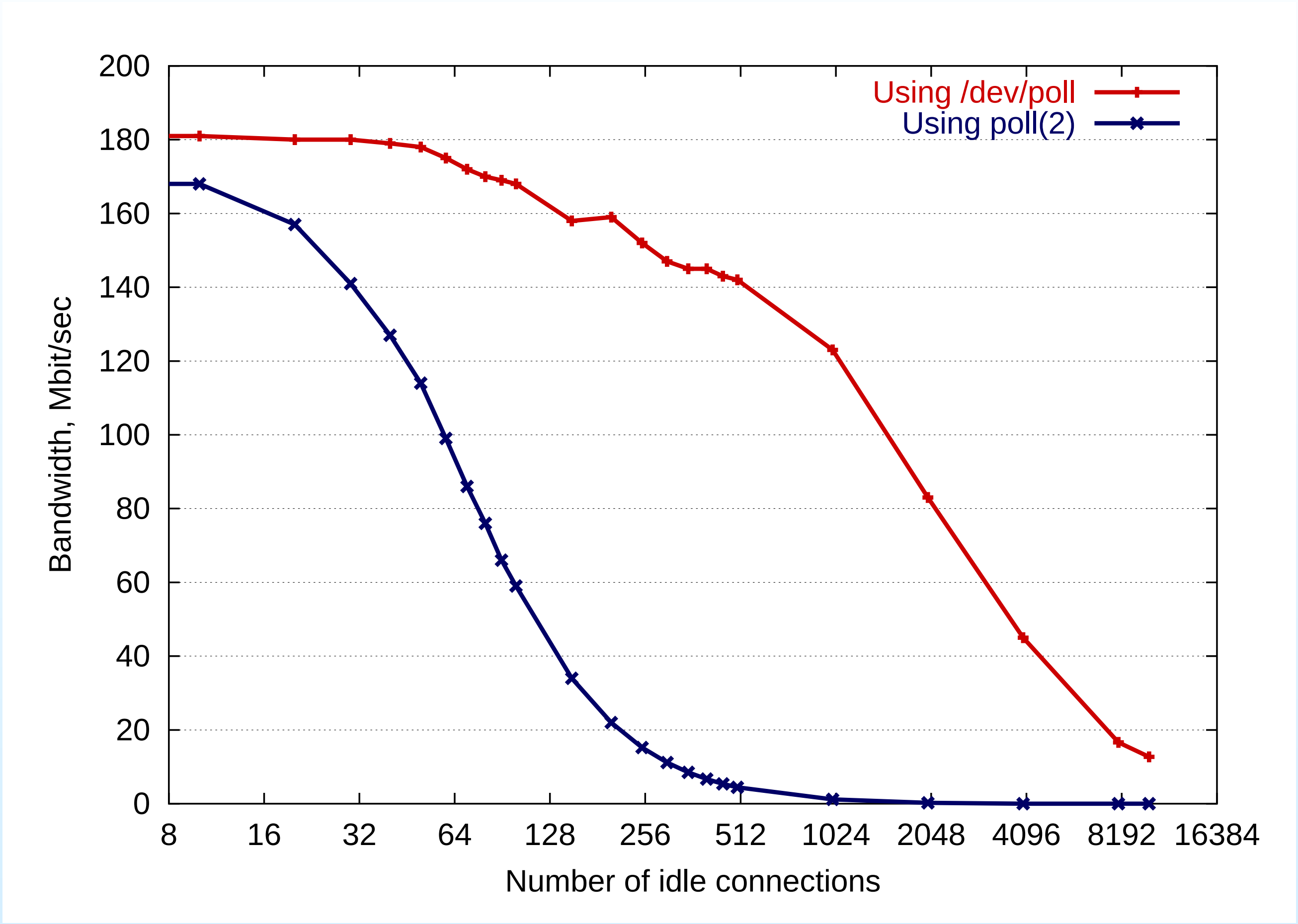
Asynchronous Sockets Performance



(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

Performance of *poll(2)* and `/dev/poll`



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

- One active connection, 1-10000 idle connections
- `/dev/poll` scales better than *poll(2)*

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

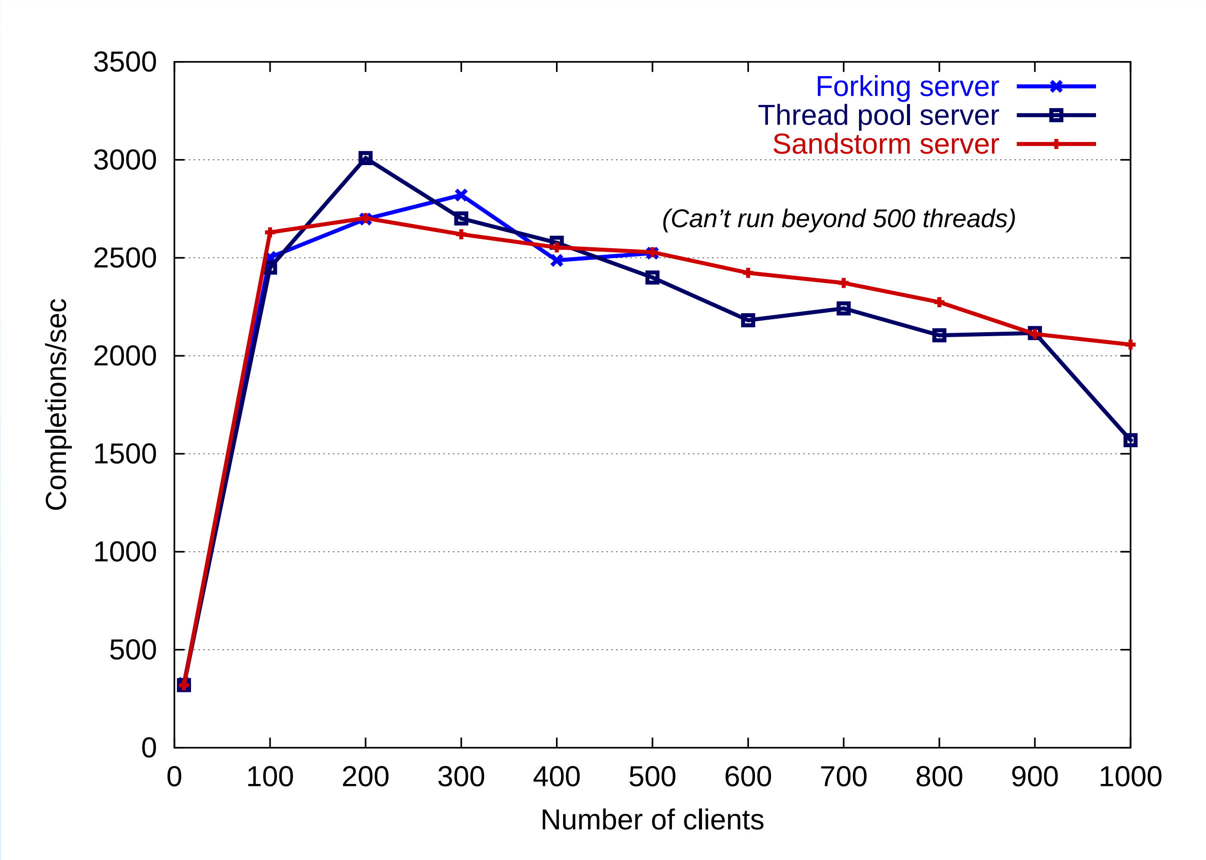
Timers

- 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

Example Application: Simple HTTP Server



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

- Return 8Kb webpage from memory, clients sleep 20 ms, 100 reqs/conn
- Sandstorm: 3 threads, nonblocking I/O
- Threadpool: 150 threads, blocking I/O
- Forking: one thread per connection, blocking I/O

Response and Connect Time

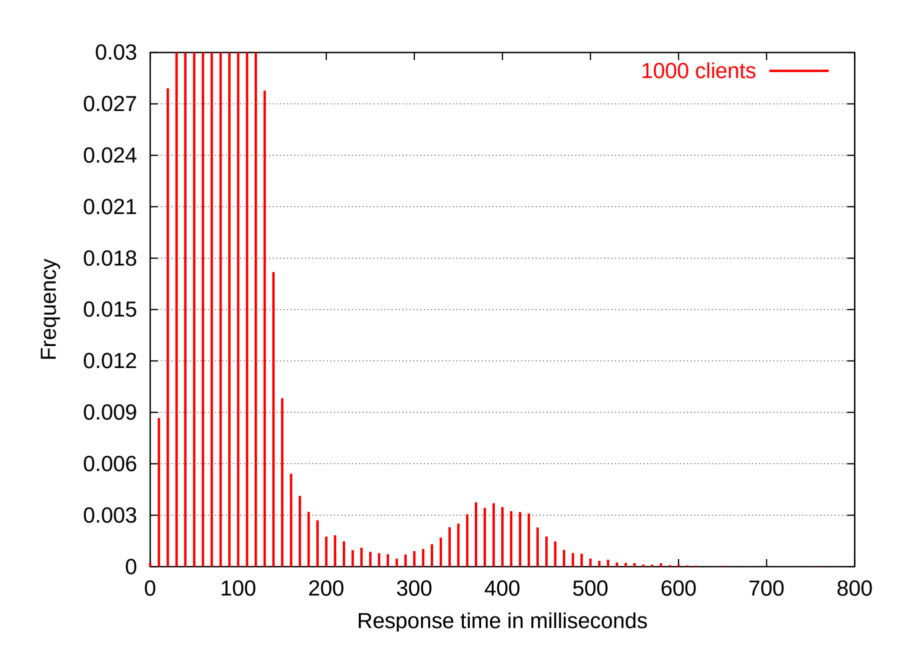
Server	*Connect time*		*Response time*	
Sandstorm	median	*420 ms*	median	*1105 ms*
(1000 connections)	max	*3116 ms*	max	*15344 ms*
Thread pool	median	*3105 ms*	median	*4570 ms*
(1000 connections)	max	*189340 ms*	max	*190766 ms*
Forking	median	*2990 ms*	median	*2920 ms*
(500 connections)	max	*45201 ms*	max	*48136 ms*

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

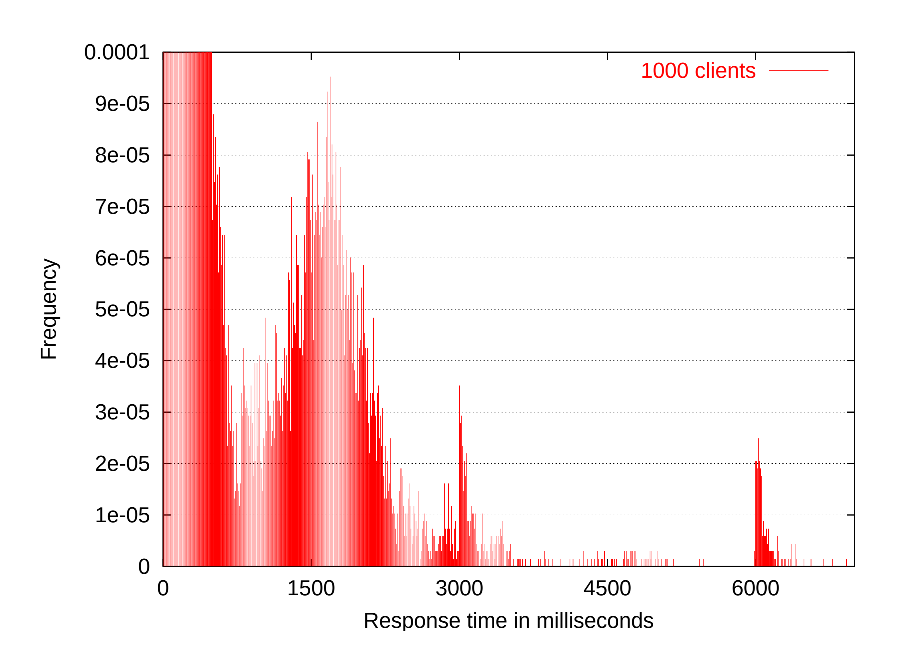
Must use alternate metrics

- ''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

(1000 connections, 4-way 500 MHz PIII, Gigabit Ethernet, Linux, IBM JDK 1.1.8)

- SEDA and threadpool servers both sustain high throughput
- But threadpool server has limited capacity: 150 threads
- Note spikes at 3000 ms intervals for threadpool server
  - ▷ *Due to TCP SYN retransmit timeout on Linux*

Another Application: Gnutella Packet Router

Goal: Explore application domains other than client/server

- Different properties and challenges

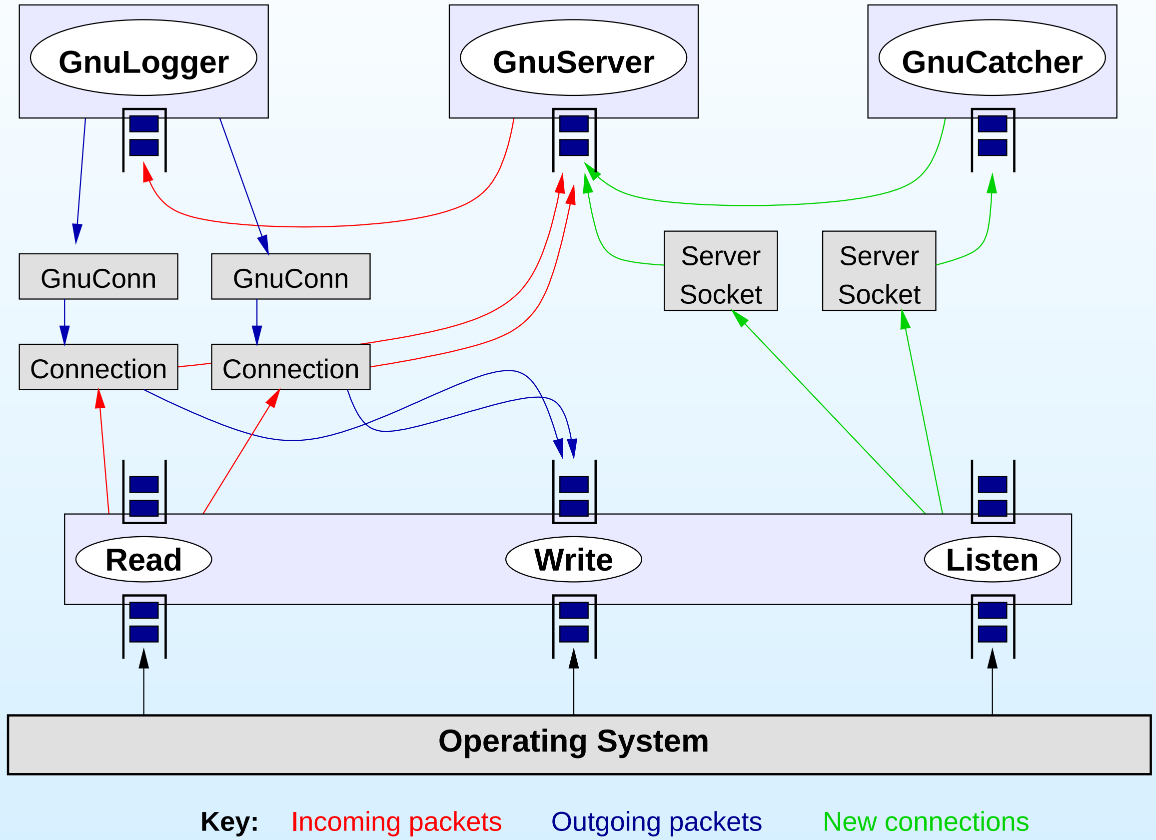
Goal: Demonstrate load conditioning

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

Gnutella basics

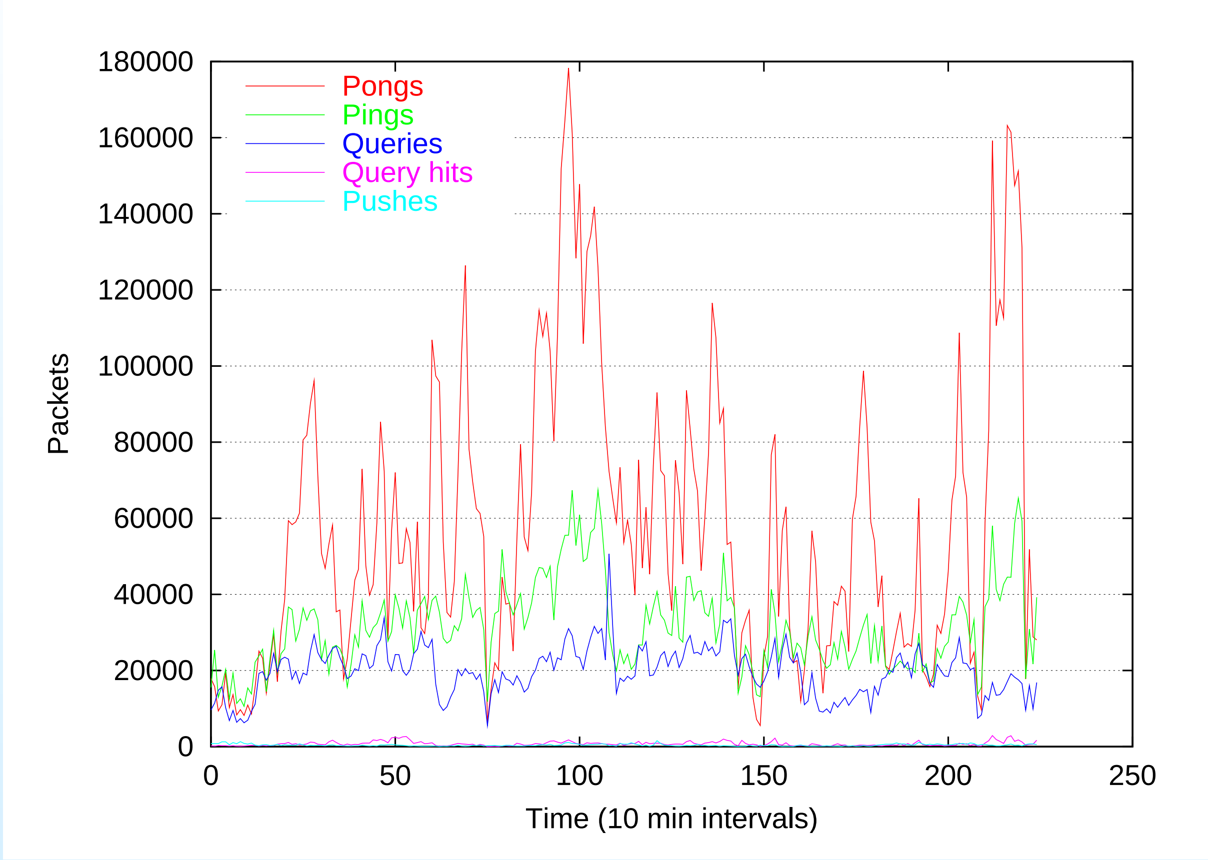
- Decentralized peer-to-peer file sharing network
- Every node exchanges messages with its neighbors
  - ▷ *ping, pong, query, queryhit, push message types*
- Direct download from host via HTTP
- Initial discovery via well-known host
- Several thousand users at any time, 10's of TBs of data

Sandstorm Gnutella Server



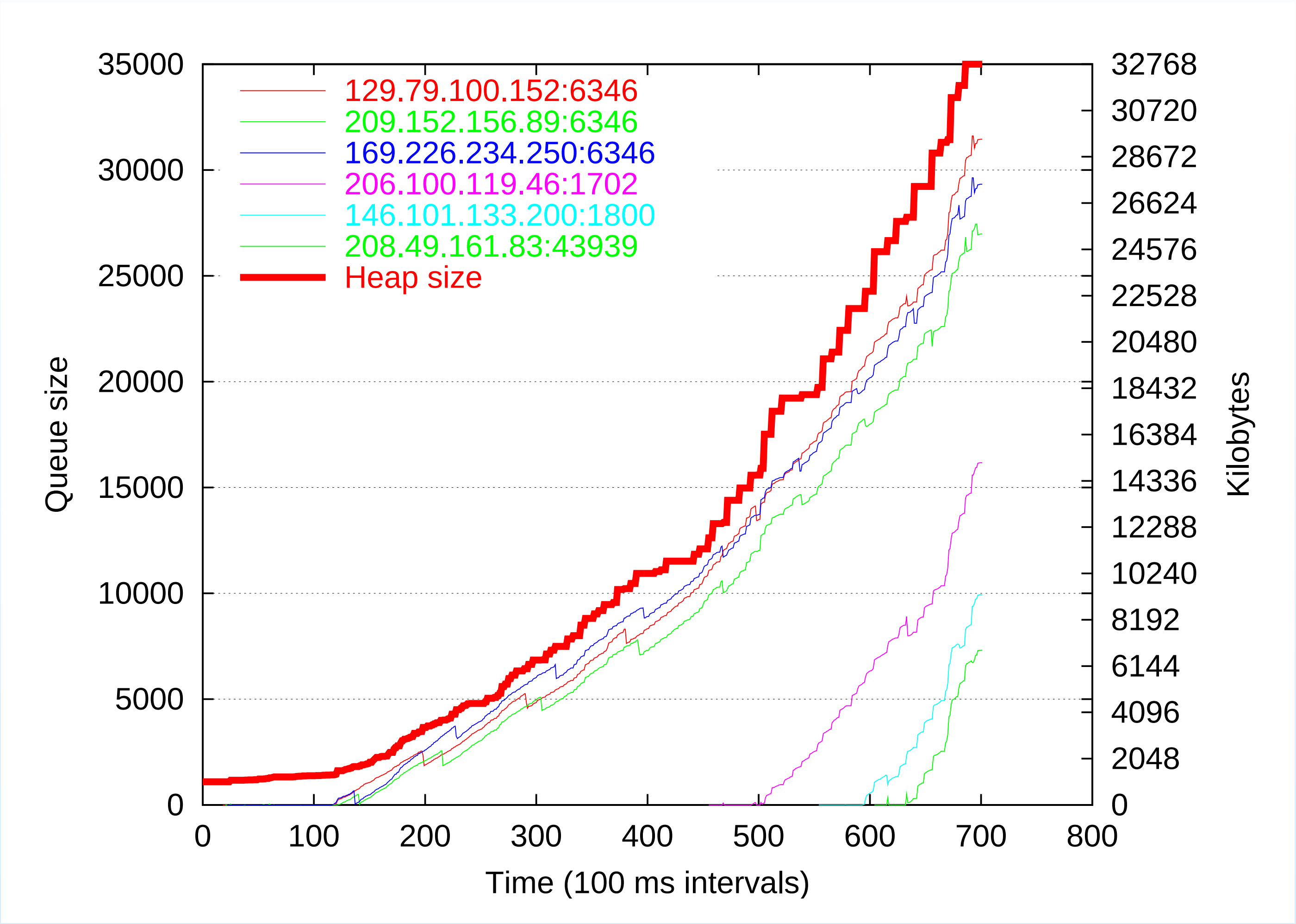
- Logger: Routes and logs packets
- Server: Parses incoming packets
- Catcher: Establishes new connections
- Gnutella Connection: Formats outgoing packets

Gnutella Packet Trace



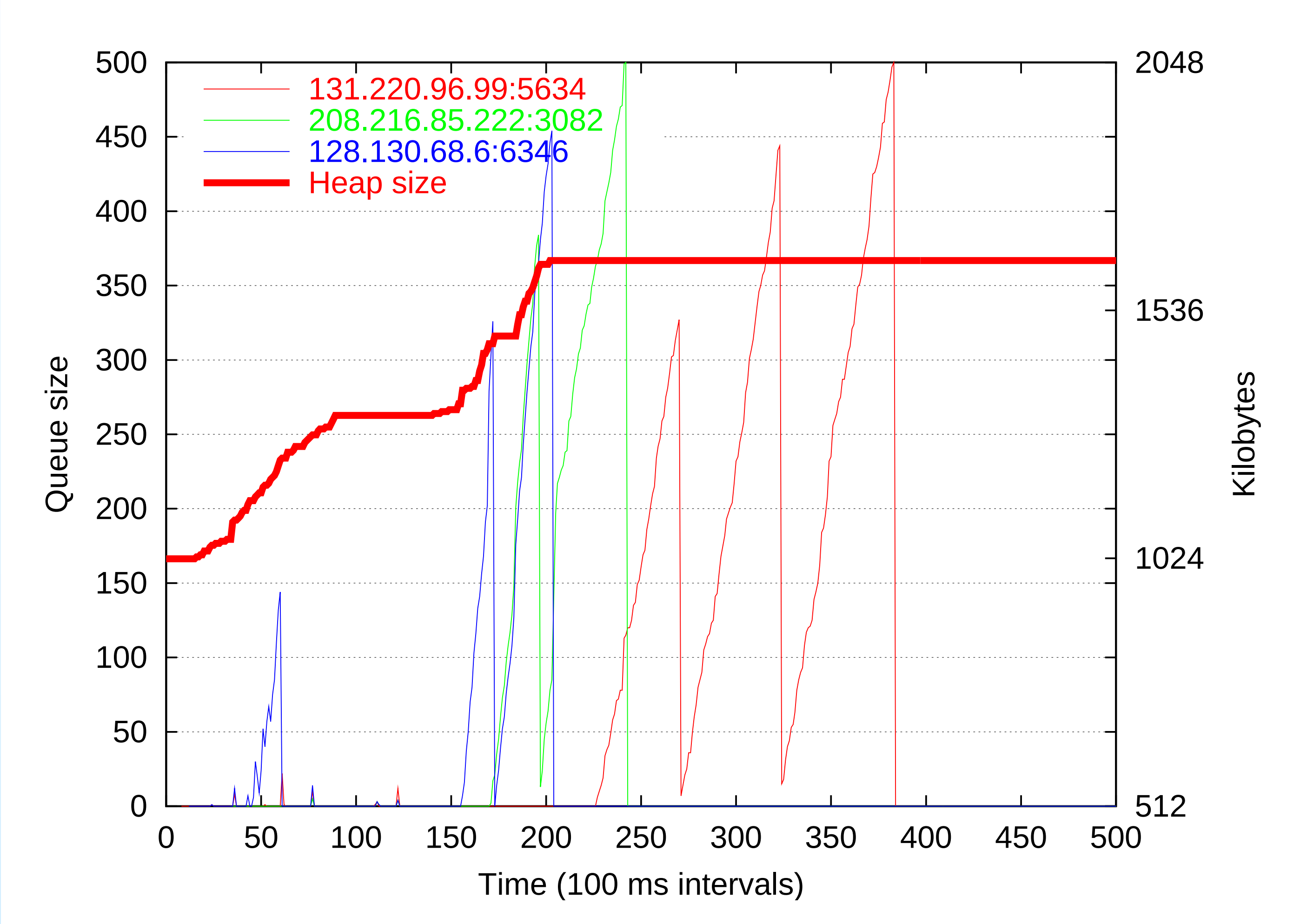
- Gathered over 37-hour period
- *24.8 million* packets, average *179.55* per sec
- *72396* connections, average 12 at any time
- Very bursty, no clear diurnal pattern

Dealing with Clogged Connections



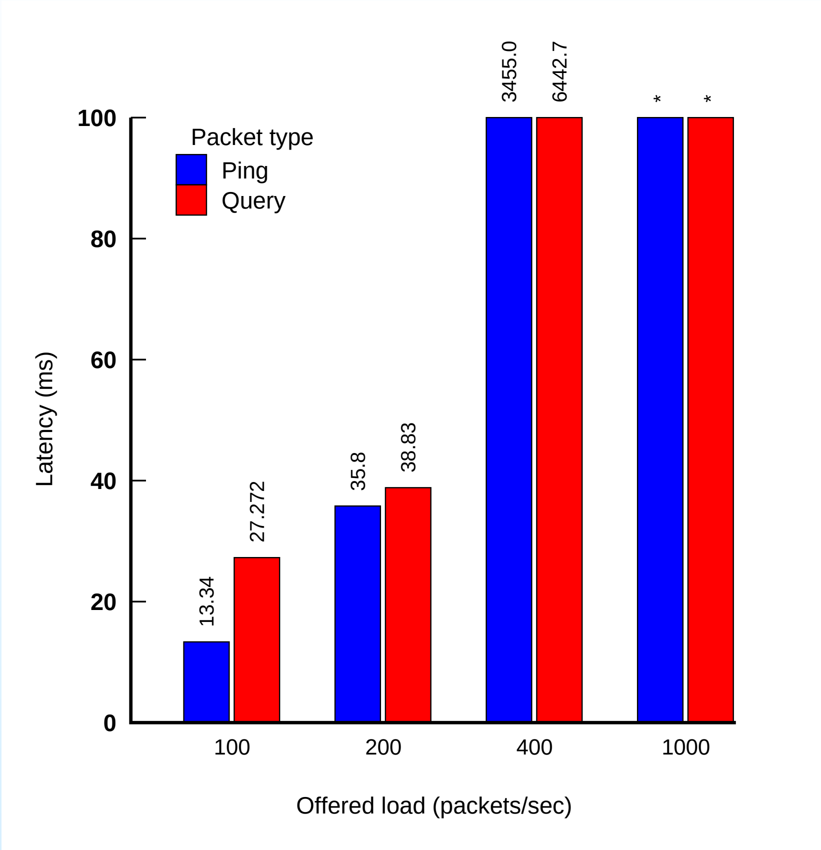
- Server would crash after a few hours
- Cause: saturated connections
  - ▷ *115 packets/sec can saturate a 28.8 modem link*

Socket Queue Thresholding



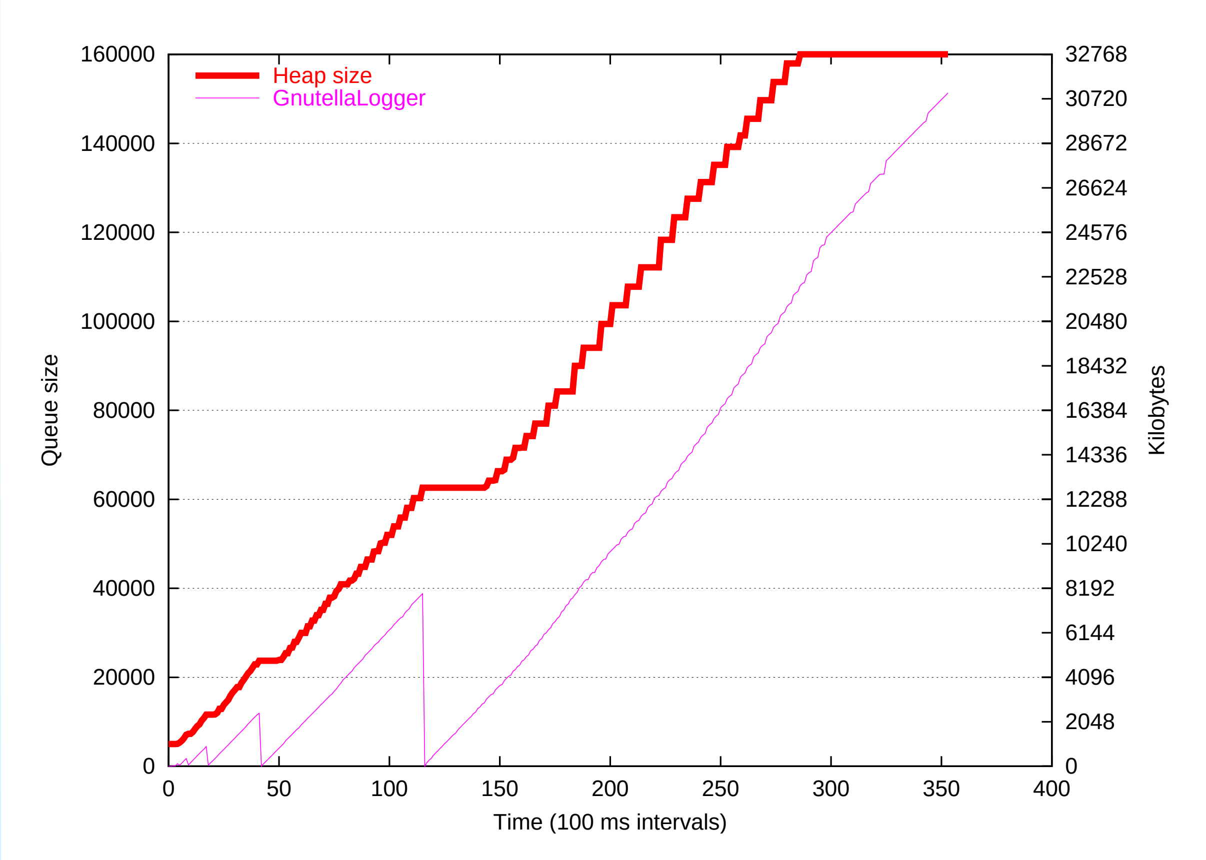
- Close connection if outgoing queue reaches threshold
- One form of load conditioning
- Many variations possible

Router Latency Under Overload Condition



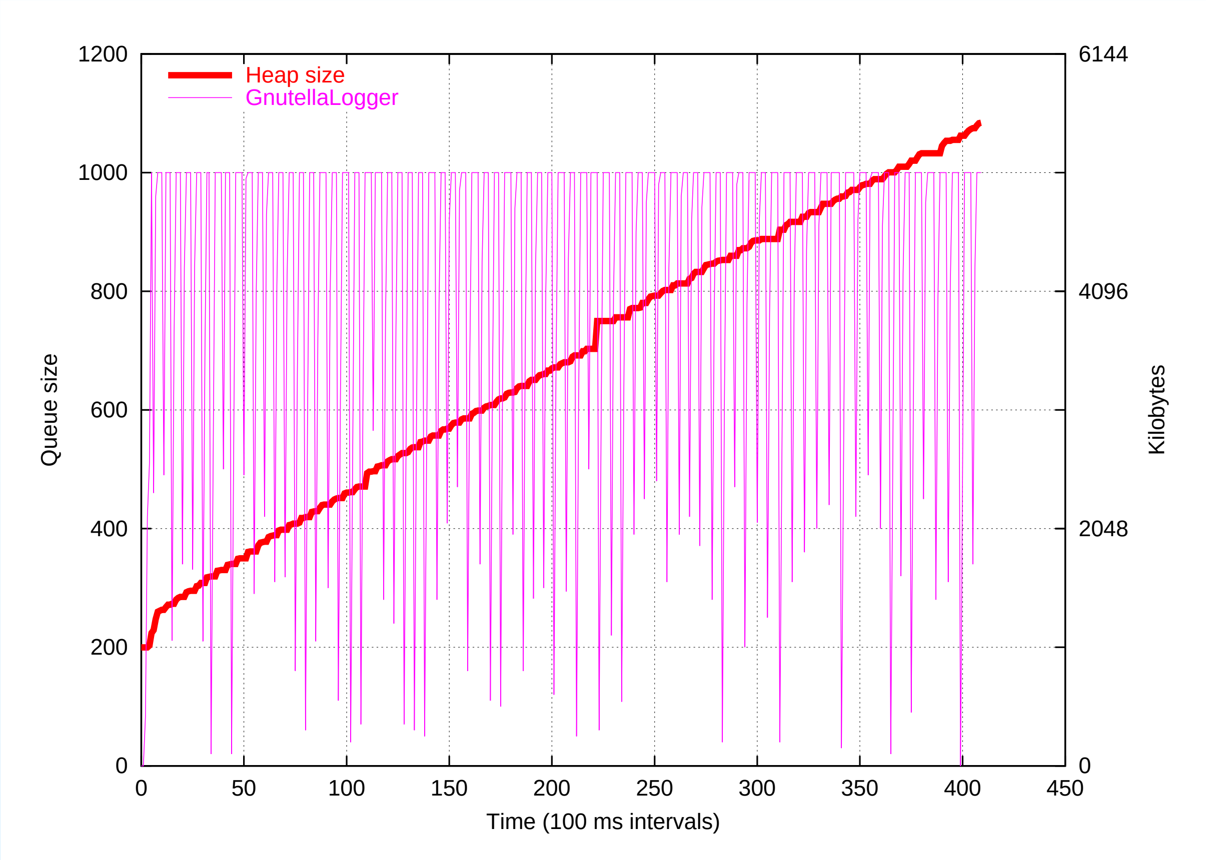
- Benchmark client generates realistic packet streams
- Introduce intentional bottleneck into server:
  - ▷ *Server-side delay of 20 ms for query messages*
  - ▷ *15% of messages are queries*
- Server crashed at 1000 packets/sec

Sandstorm Profile of Overloaded Server



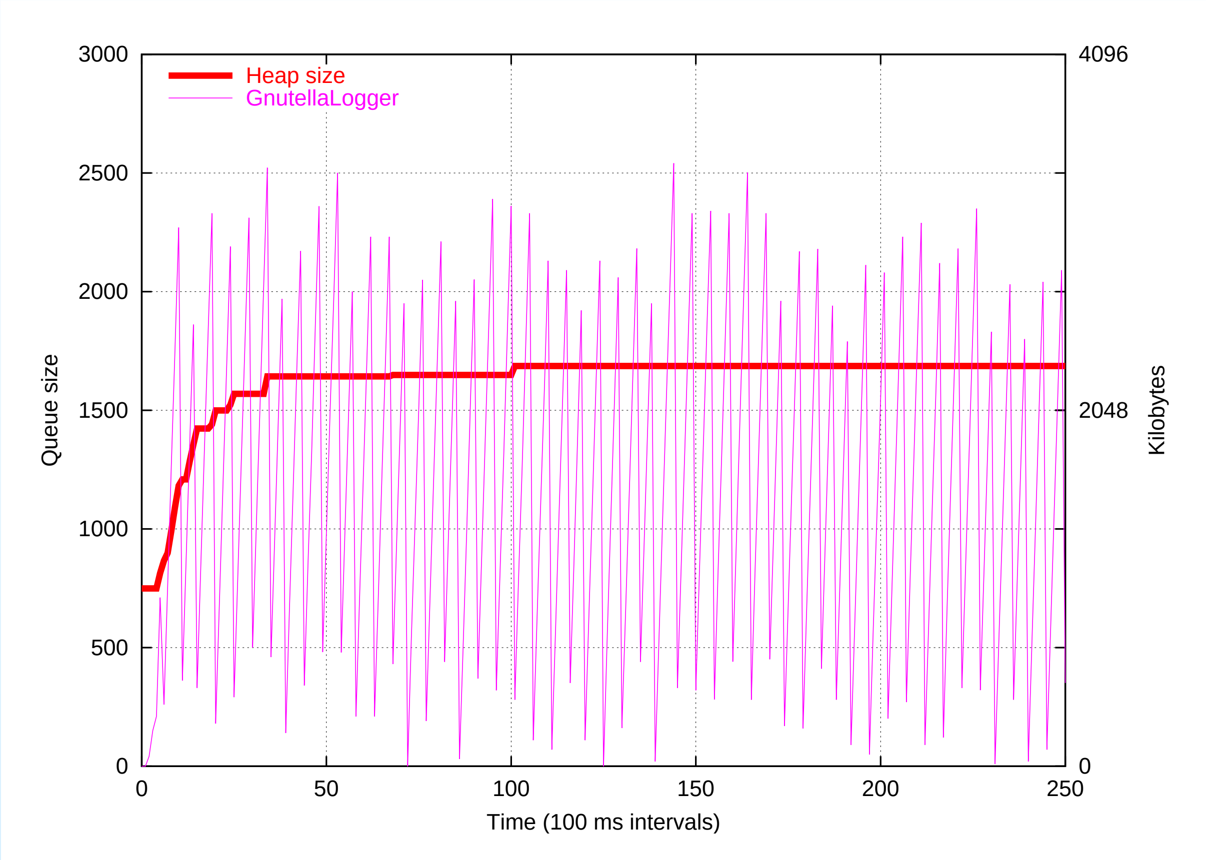
- Offered load of 1000 packets/sec
  - *Query message delay of 20 ms*
- Clearly indicates *GnutellaLogger* stage as bottleneck

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

Sandstorm Thread Governor

Dynamically adjust number of threads servicing a stage

- 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

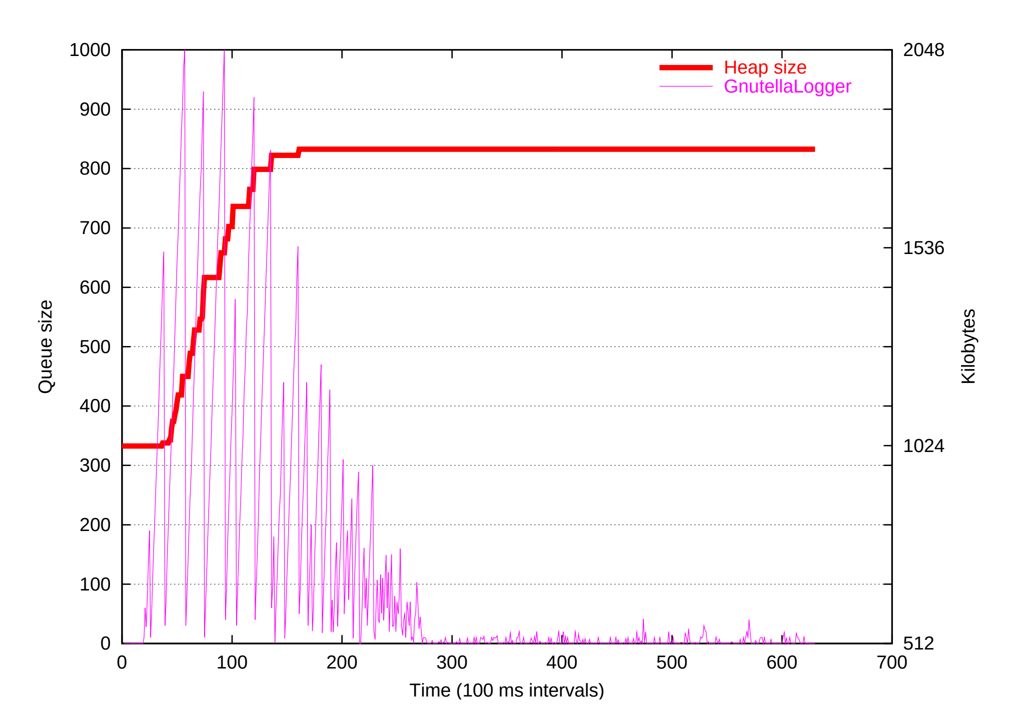
$$n = \lambda p L = (1000)(0.15)(20 \text{ ms}) = 3 \text{ threads}$$

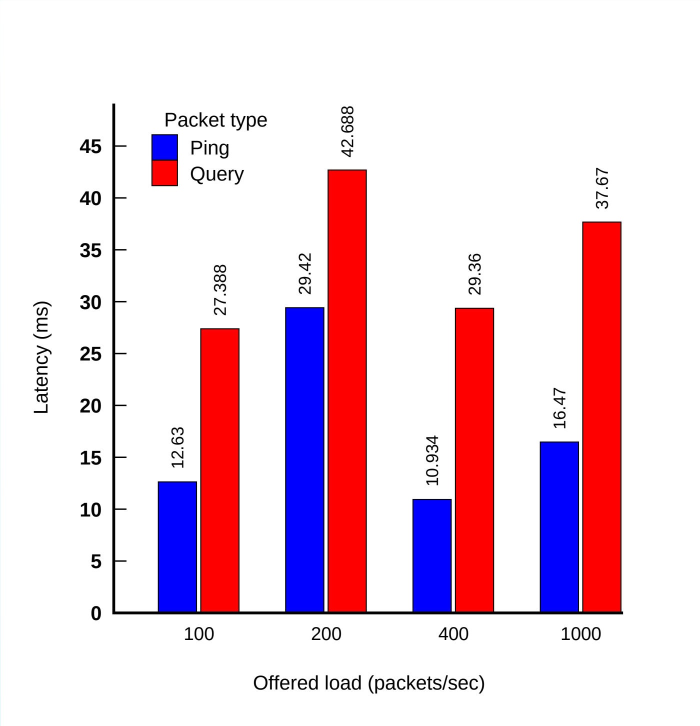
  - ▷ *Unfortunately, can't determine a priori*

Controller based on observation of queue lengths

- Sample queue lengths every 2 sec
- Add a thread when queue reaches threshold
  - ▷ *Up to some maximum number of threads*

Sandstorm Thread Governor





- Add thread when event queue reaches threshold of 1000
  - ▷ *2 threads added to GnutellaLogger stage*
  - ▷ *Matches theoretical result of 3 total threads*
- Response time stabilizes
  - ▷ *Higher for 200 packets/sec, since no threads added*

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

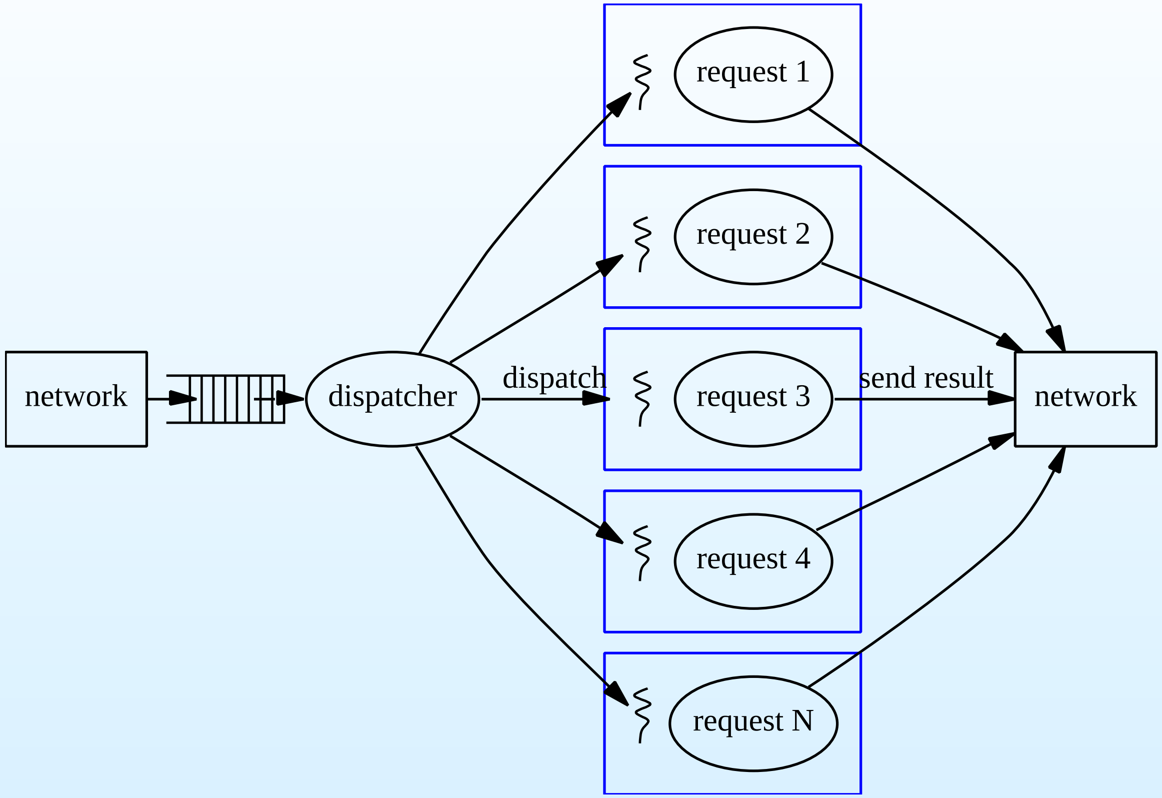
- *Sandstorm* service platform
- Application scalability and load conditioning

For more information

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

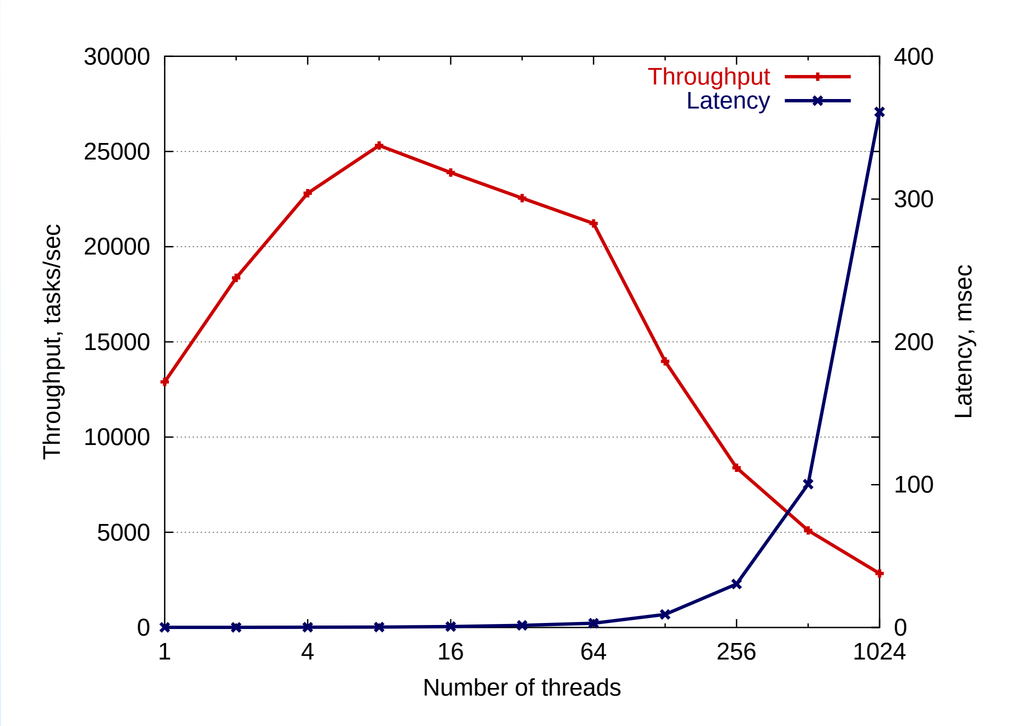
Backup Slides Follow

Thread-Based Concurrency



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

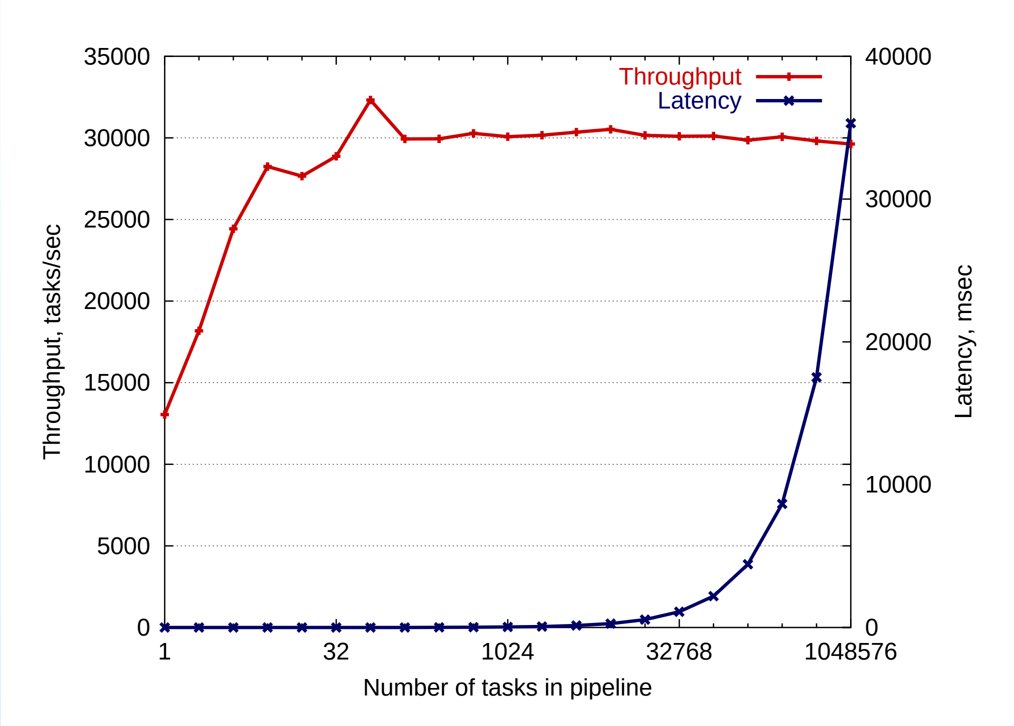
Problems with Threads



(930 MHz Pentium III, Linux 2.2.14, read 8Kb cached file)

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

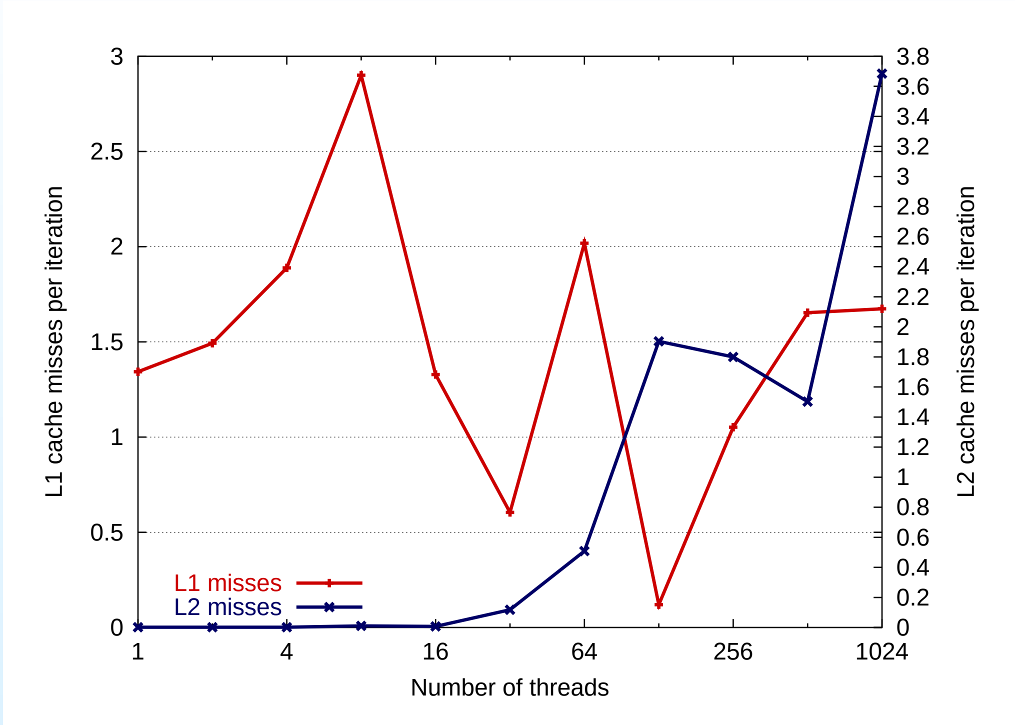
Event-Driven Server Performance



(930 MHz Pentium III, Linux 2.2.14, read 8Kb cached file)

- Constant throughput up to pipeline size of 2^20 (1 MB)

Cache Misses, Threaded Server

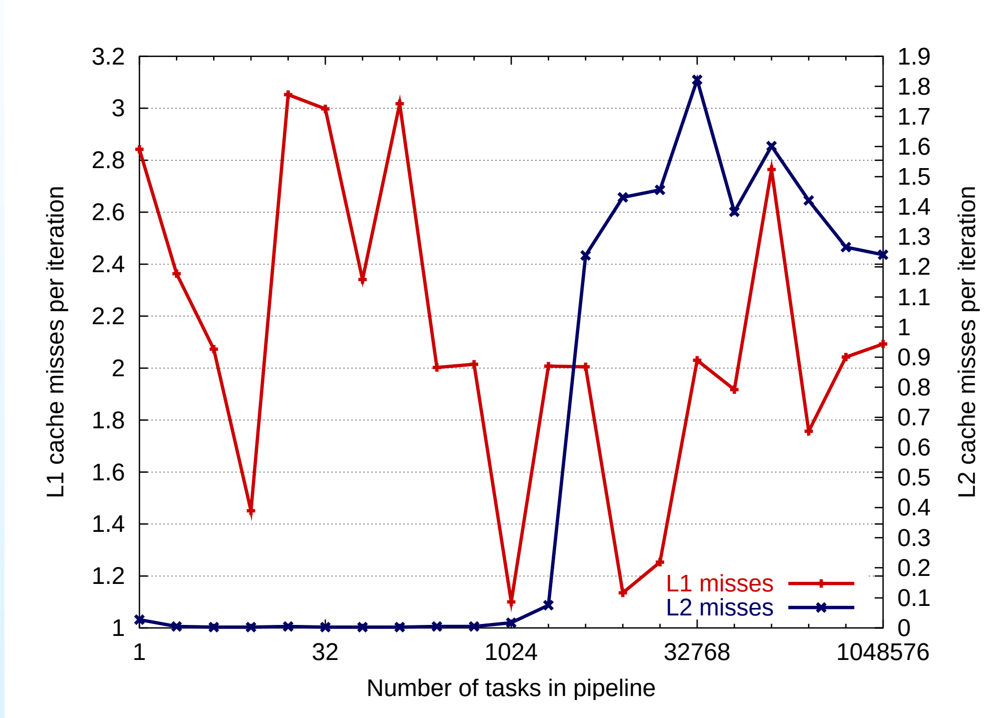


(930 MHz Pentium III, 16 KB L1, 256 KB L2)

Pentium hardware counters used to get statistics

- Both user and OS-level cache misses counted
- L1 misses nearly constant
- L2 misses increase as more threads added

Cache Misses, Event-Driven Server



(930 MHz Pentium III, 16 KB L1, 256 KB L2)

- L1 and L2 misses nearly constant
- One L2 miss per iteration when event descriptors don't fit in cache
    - ▷ *Event descriptor is 60 bytes*
    - ▷ *At pipeline size of 8192, these no longer fit in L2*