Building Dependable Internet Services

(Or, what I learned doing my thesis)

Matt Welsh

Intel Research Berkeley
and Harvard University

mdw@cs.berkeley.edu

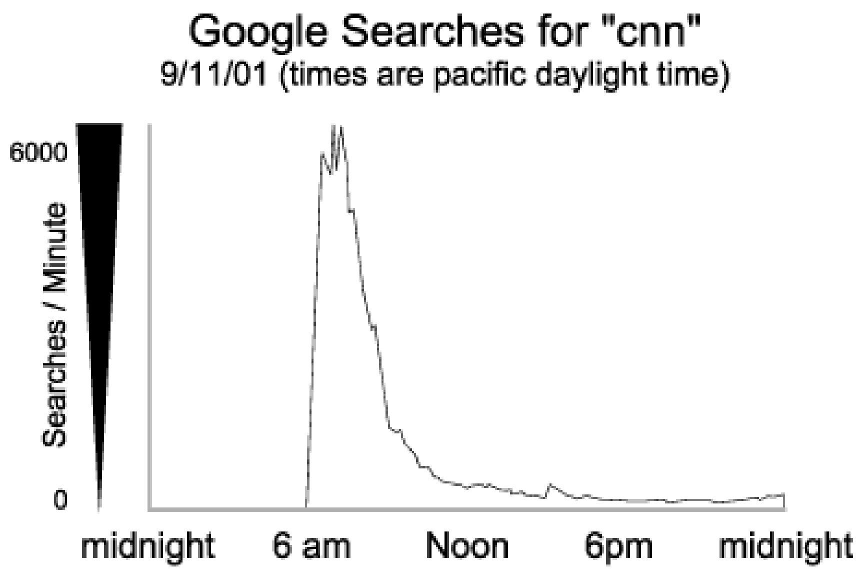
The Problem: Overload in the Internet

Overload is an inevitable aspect of systems connected to the Internet

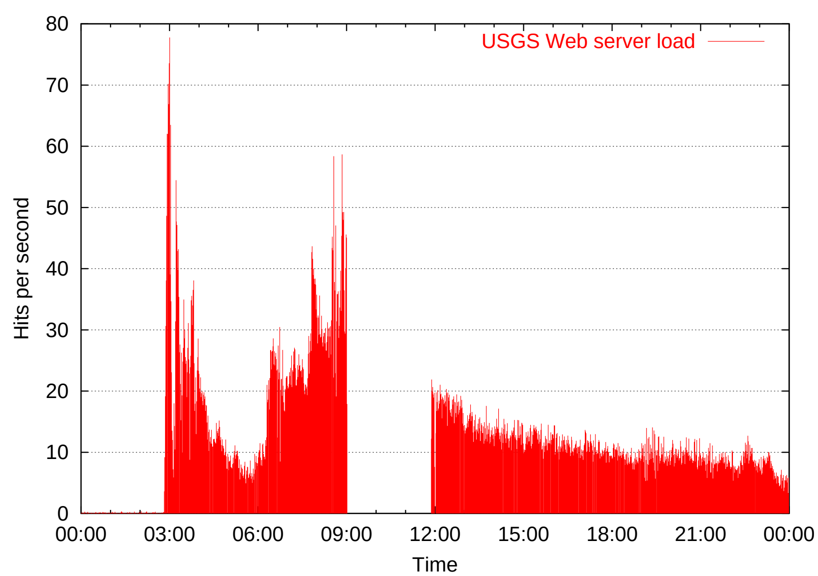
- (Approximately) infinite user populations
- Large correllation of user demand (e.g., flash crowds)
- Peak load can be orders of magnitude greater than average

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



USGS site load after earthquake



Overload management is hard

Throwing more resources at the problem does not work

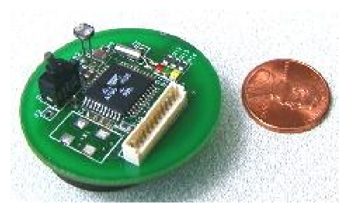
- Can't overprovision when load spikes are 100x or more

Not all Internet-connected systems are in big data centers

- Peer-to-peer systems: Slow PCs at home
- Edge cache servers and CDNs: Akamai, Inktomi
- Global collaborative storage systems: OceanStore, PAST
- Sensor networks: Small number of connected base stations

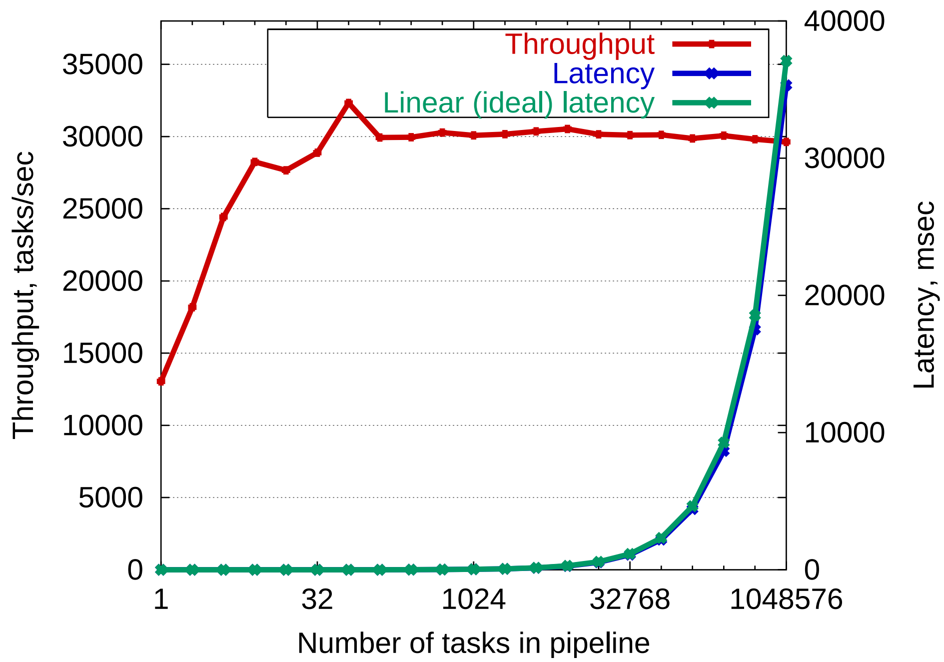
Project JXTA

oceanstore

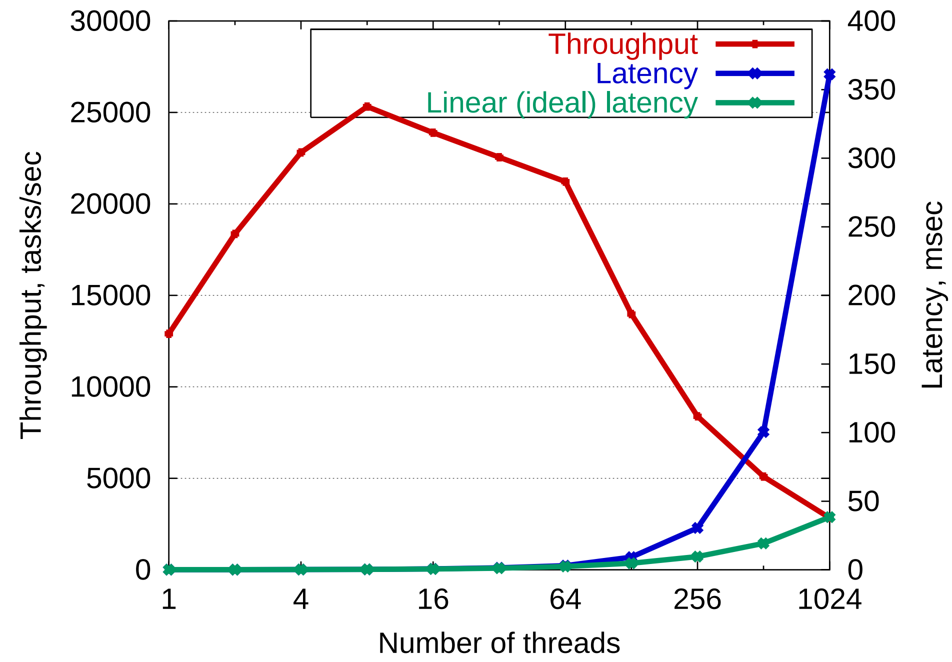


Lime wire

One axis of dependability: Well-conditioned behavior



Well-conditioned



Poorly-conditioned

Service should behave like a *pipeline*

- Throughput saturates at some load
- Response times grow linearly (or at least predictably)

Typical case: Uncontrolled degradation

- Overload causes throughput to *drop* dramatically
- Response times grow without bound
- TCP connection backoff exacerbates the problem

What's deployed now

Apache and IIS are the most popular Web servers

- Backed by wide range of databases and application servers
- Process- or thread-driven concurrency (doesn't scale)
- Simple load management policies
    - ▷ *e.g., bound on number of simultaneous connections*

Scaling through replication

- Grow a big cluster (or several big clusters)
- Sometimes load-share across different apps/sites
- Smart switches for load-balancing and request redirection
- Individual nodes still experience huge variations in load

Akamai (& friends) for static content

- Distribute content widely, but might touch central server
- Efforts underway to host dynamic scripts
    - ▷ *These often require centralized database access*
    - ▷ *Not often cacheable*

Prior work in overload control

Often based on static resource limits

- Fixed limits on number of clients, CPU utilization, listen queue thresholds
- Can underutilize resources (if limits set too low), or lead to oversaturation (if limits too high)
- No connection to user-perceived performance

Static page loads or simple performance models

- e.g., Assume linear overhead in size of Web page
- Can't account for comple, dynamic services
  - ▷ *Scripts, database/app server access, SSL, etc.*

Many studied only under simulation

- Fails to capture full complexity of real services

Prequel - the Berkeley Ninja project

Cluster-based platform for scalable, fault-tolerant services

- Much similarity to WebSphere/WebLogic cluster J2EE platforms

Incorporated an “asynchronous I/O core”

- Part of Steve Gribble’s thesis work
- Avoid dedicating a thread to each request/task
- Blocking operations become split-phase

Based on unstructured upcall mechanism

- Requests flow through chain of FSMs connected with upcalls
- Upcalls can be instantiated as queues
- Results in spaghetti code: Hard to follow control flow!
- No direct support for overload management

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

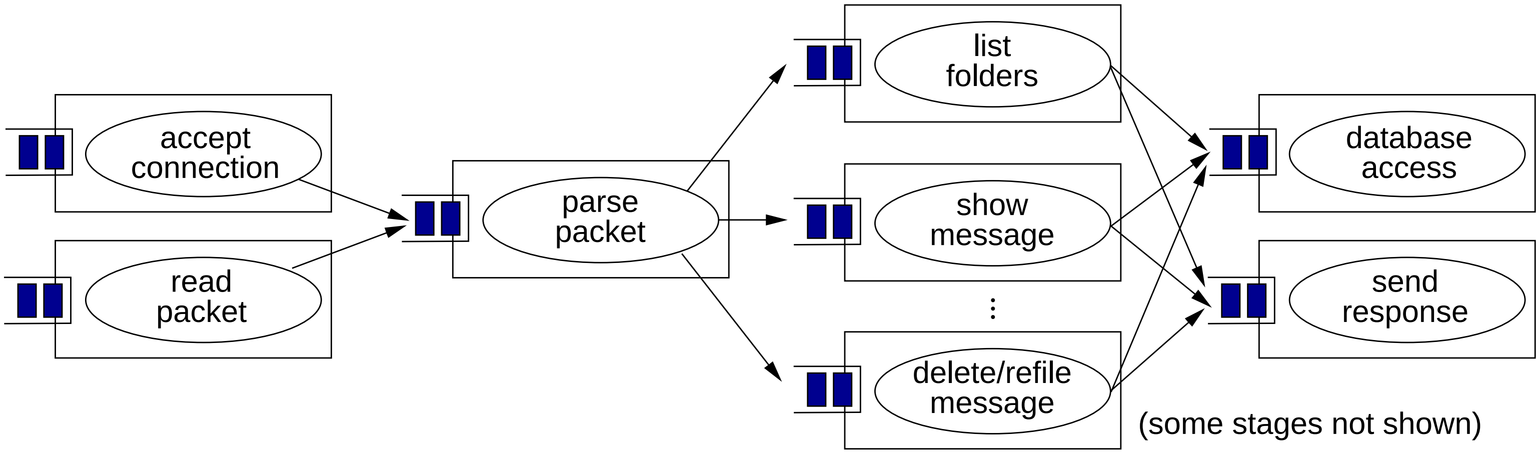
Self-tuning resource management

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

Fine-grained admission control

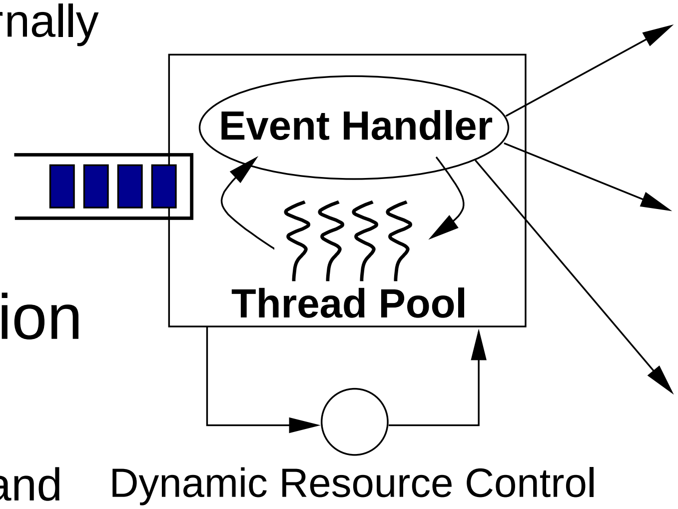
- Control flow of requests **through** service
- Smooth bursts and automatically detect resource bottlenecks

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

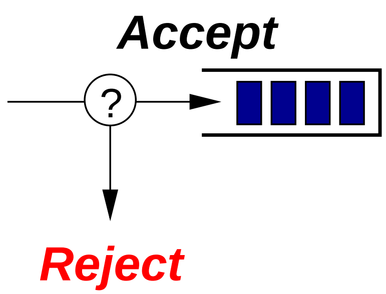
- Small number of threads per stage
- Dynamic control grows/shrinks thread pools with demand

Applications implement simple *event handler* interface

- 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

Fundamentally: 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`)

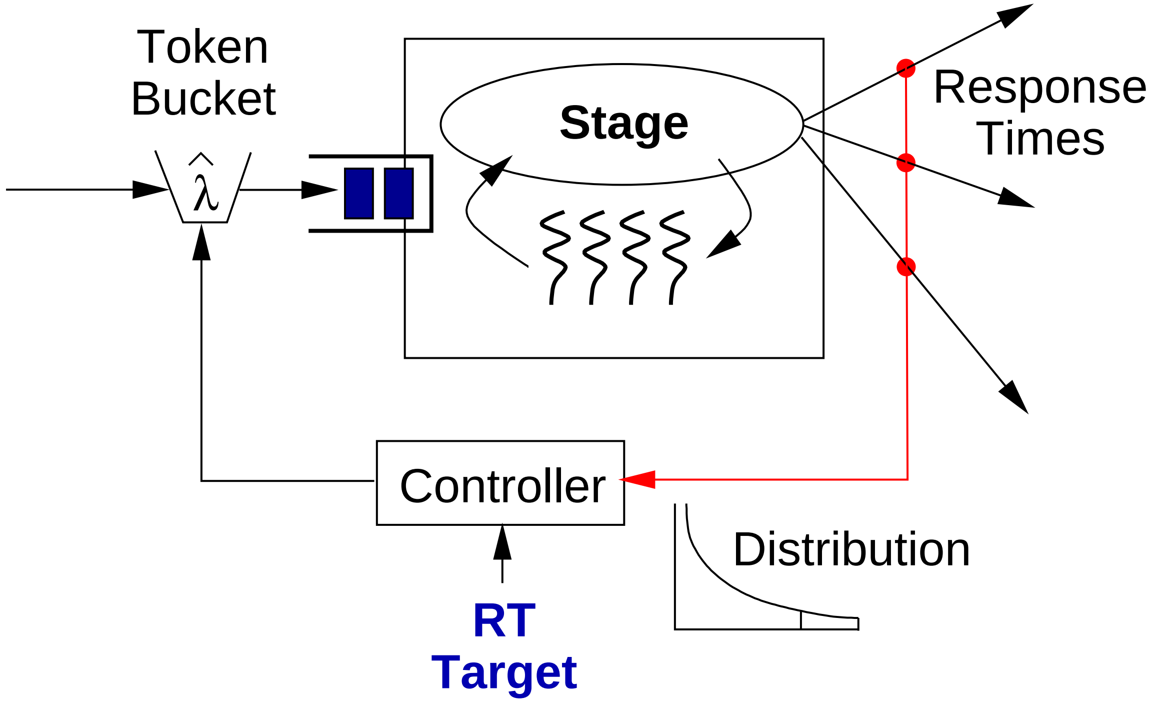
- Can couple with front-end load balancing across server farm

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



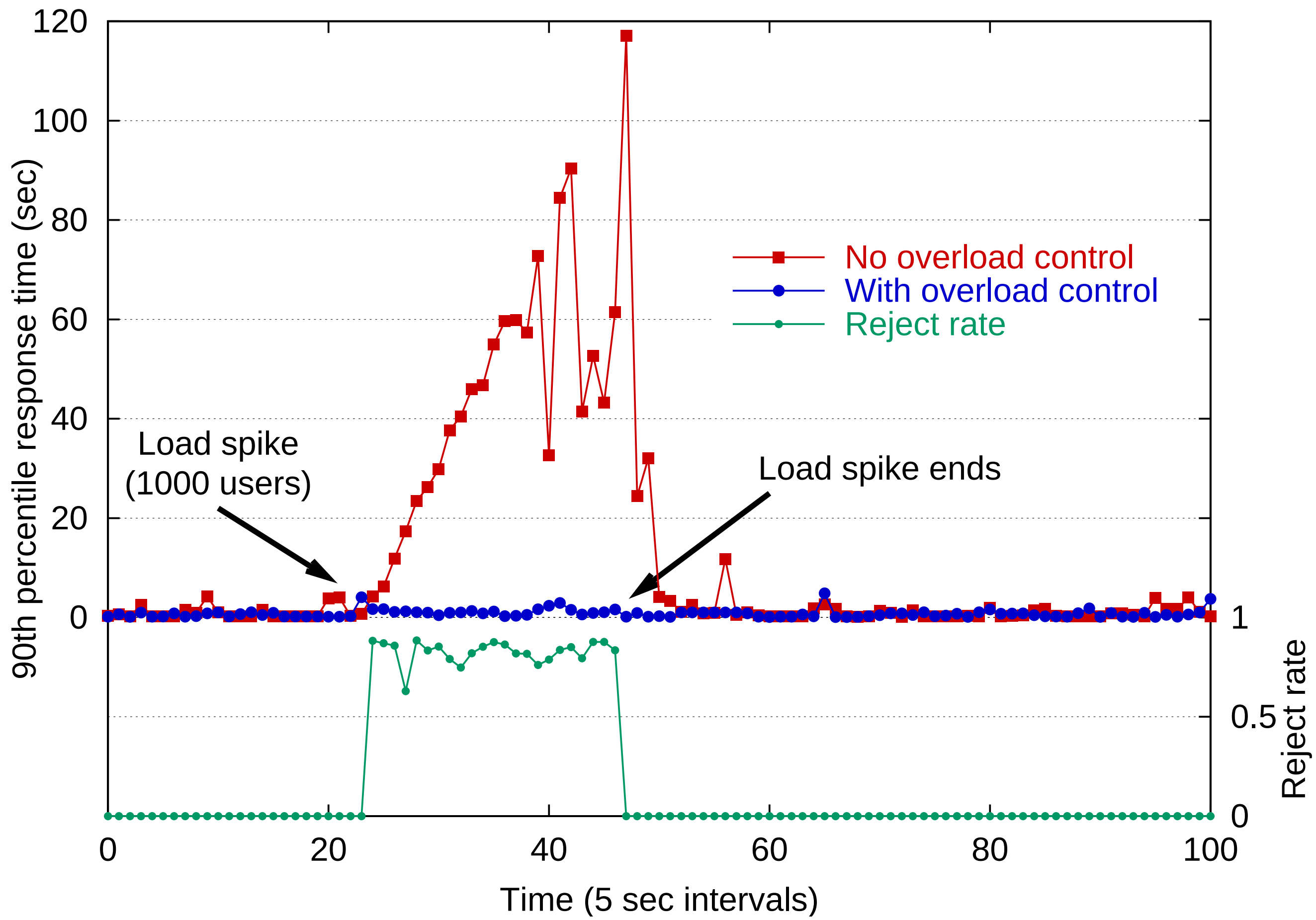
Adaptive admission control at each stage

- Target metric: Bound *90th percentile response time*
- Measure stage latency, throttle incoming event rate

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

Overload prevention during a massive load spike



Sudden spike of 1000 users on SEDA-based email service

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

Overload controller has no knowledge of the service!

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 procesing “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 hacks, but very effective (and clean!)

User-level vs. kernel-level resource management

SEDA is a **user-level** solution: no kernel modifications!

- 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?

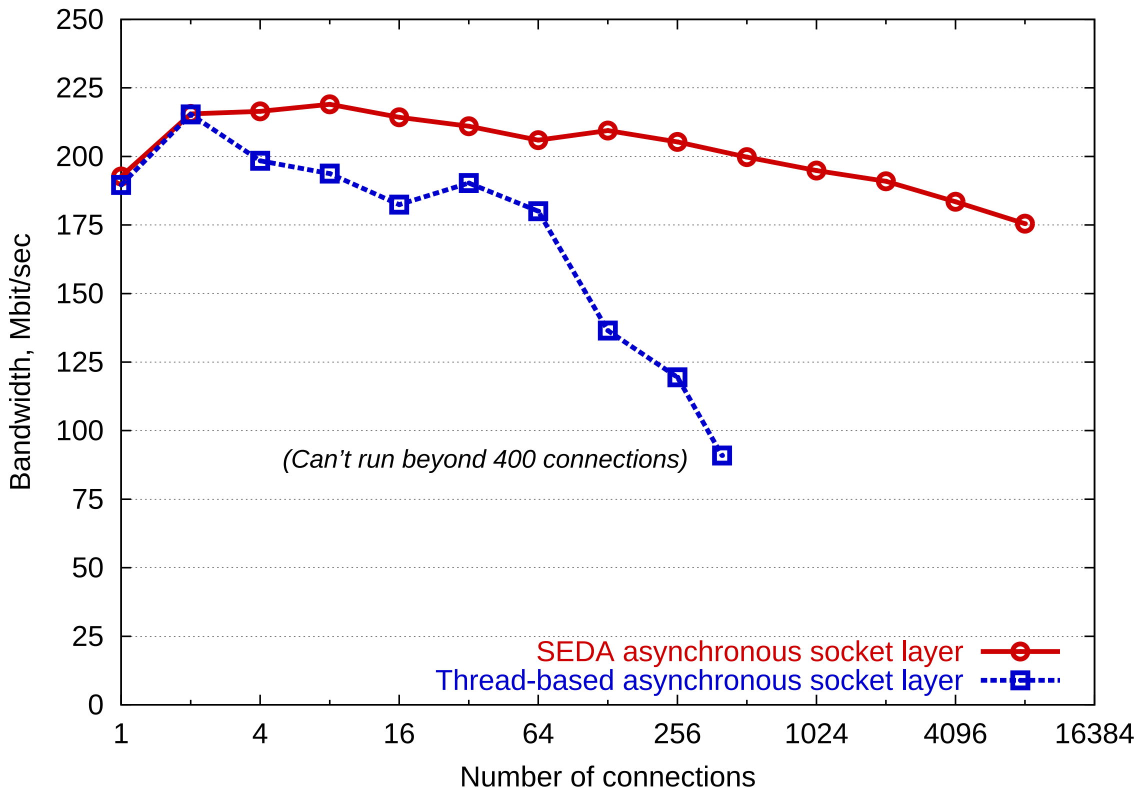
- Scalable I/O primitives: remove emphasis on blocking ops
- SEDA stage-aware scheduling algorithm?
- Greater exposure of performance monitors and knobs
  - ▷ *Double-edged sword: facilitates feedback and control, but awfully complex*

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

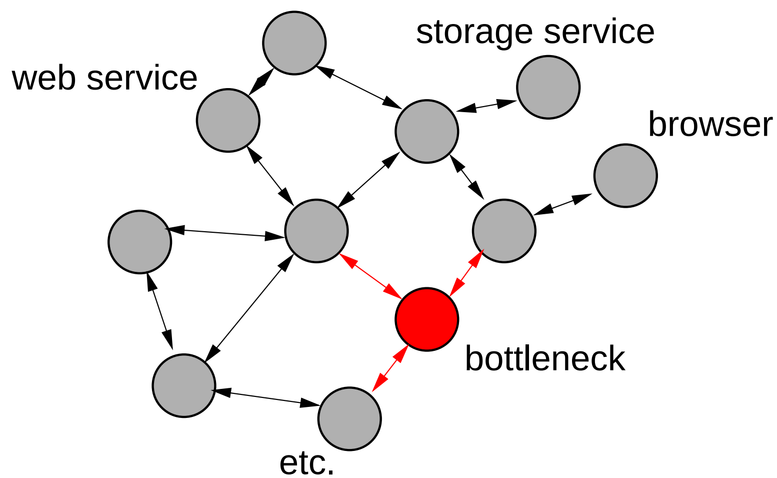


- Many kernels not designed/configured for high I/O concurrency

Distributed programming models and protocols

Distributed computing models 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?
- Not accepting TCP connections is the **wrong** way to manage overload
- Single bottleneck in large distributed system causes cascading failure in network



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*

Control theoretic resource management

Increasing amount of theoretical and applied work in this area

- Control theory based on 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*

Difficult to **prove** anything about resulting system

- Much control theory based on linear models
  - ▷ *Real software systems highly nonlinear*

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

Distributed load management

Translating overload control to clusters

- Naive approach: Run SEDA on each node, load-balance across nodes
- Opens up options for admission control
  - ▷ *e.g., Redirect request to less-loaded node*
- Longer delays and more uncertainty in the feedback loop

Resource and application sharing

- Smarter resource balancing across competing apps [Roscoe, Chase, etc.]
- Introduces interesting dependencies
  - ▷ *Cartoon Network overload on 9/11 caused by CNN*

What about centralized resources?

- e.g., Single back-end database hosting entire site
- Web- or application-tier can condition load on database

So what did I really learn?

Focusing on *robustness* rather than *raw performance* changes your world-view

- 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*

We need better formalisms for studying complex systems

- Too many "seat of the pants" algorithm designs
- Would be nice to prove the stability of our approach to overload control
- Simulations and abstractions are tempting but potentially dangerous

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