Overload Management as a Fundamental Service Design Primitive

Matt Welsh

UC Berkeley Computer Science Division

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 correlation 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
Outline

Why overload management is hard

Why current OS and programming models don’t help

The case for feedback-driven control

SEDA: System architecture for well-conditioned services

Some examples:

- Adaptive admission control
- Service degradation under overload
- Class-based service differentiation

Future research directions
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
Resource management and overload exposure

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”

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

![Diagram showing network with a bottleneck]
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 need for dynamic overload control

Classic approach: *a priori* resource limits

- e.g., Bounding number of TCP connections or threads
- Static resource shares (e.g., Process P gets 10% of the CPU)

Problems with static resource containment

- Static “knobs” hard to set
- May lead to underutilization
- Can still cause overload if limits set too high

We argue for *feedback driven control*

- Actively observe performance and adjust resource usage
- Maintain high utilization
- Much more flexible than static allocation
- Similar to measurement-based admission control in networks

▷ *Perform AC based on measured load, rather than impose static limits*
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
SEDA Architectural Features

Efficient event-driven design

- Small number of threads *per stage*, not thread per request
- Decomposition into stages simplifies application development

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.*

Dynamic control for self-tuning resource management

- System observes application performance and tunes runtime parameters
- e.g., Control number of threads per stage, number of events processed in one batch
- Adaptive admission control at each stage to prevent overload
SEDA Programming Model

Stages export single method: `handleRequests()`

- Process a **batch** of requests - can amortize work
- Request processing may be multithreaded
  - Avoid shared state and locks
  - Must take care when processing requests out-of-order

Overload management is explicit!

```java
public void handleRequests(request_t batch[]) {
    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.
        }
    }
}
```
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:

Mata Welsh, UC Berkeley
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!
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
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 × target RT, add $f(\text{err})$ to rate
  - If 90th RT > target RT, divide rate by 1.2
Sudden load spike of 1000 users hitting Arashi 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!
Overload management using 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

Matt Welsh, UC Berkeley
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
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 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
Future Directions

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
Future Directions 2

New system designs for detecting and preventing overload

- Tradeoff between transparency and application specificity?
- Resource virtualization is tempting and dangerous

Models for **practical** large-scale distributed systems that take overload into account

- Death to RPC
- Influence on future J2EE/.NET/SOAP/etc. framework

Design issues for feedback and self-tuning in complex systems

- How to avoid “tuning the tuner”
- How much (formal) complexity is needed?
  - *Lots of hairy control theory is possible, but useful?*
- Distributed control?
- Interaction between different controllers?
Summary and Conclusions

Overload management is critical for Internet-connected systems

- Flash crowds, load spikes, and denial-of-service attacks
- Especially important as novel service designs emerge
- e.g., Complex, interdependent “Web services”

Existing service designs do not facilitate overload management

- Typically naive about performance or load conditions
- Simple, static knobs (e.g., maximum number of connections)
- Distributed system primitives (e.g., RPC) fail to expose load

SEDA makes load management a first-class design primitive

- Design for scalability: efficient event-driven concurrency
- Self-tuning resource management: feedback-driven control
- Prevent overload: Per-stage adaptive admission control

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