SEDA: An Architecture for Scalable, Well-Conditioned Internet Services

Matt Welsh, David Culler, and Eric Brewer

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

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

SOSP’01, Lake Louise, Canada
Internet Services Today

Massive concurrency demands
- Yahoo: 1.2 billion+ pageviews/day
- AOL web caches: 10 billion hits/day

Load spikes are inevitable (the "Slashdot Effect")
- Peak load is orders of magnitude greater than average
- Traffic on September 11, 2001 overloaded many news sites
- Load spikes occur exactly when the service is most valuable!
  ▶ In this regime, overprovisioning is infeasible

Increasingly dynamic
- Days of the "static Web" are over
- Majority of services based on dynamic content:
  ▶ e-Commerce, stock trading, driving directions, etc.
- Service logic evolves rapidly
  ▶ Increases complexity of engineering and deployment
Problem Statement

Supporting massive concurrency is hard

- Threads/processes designed for timesharing
  - High overheads and memory footprint
- 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
- Static resource containment is inflexible
  - How to set a priori resource limits for widely fluctuating loads?
- Load management demands a feedback loop

Dynamics of services exaggerate these problems

- Much work on performance/robustness for specific services
  - e.g., Fast, event-driven Web servers
- As services become more dynamic, this engineering burden is excessive
- Replication alone does not solve the load management problem
Proposal: The Staged Event-Driven Architecture

SEDA: A new architecture for Internet services

- A general-purpose framework for high concurrency and load conditioning
- Decomposes applications into stages separated by queues
- Adopt a structured approach to event-driven concurrency

Enable load conditioning

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

Dynamic control for self-tuning resource management

- System observes application performance and tunes runtime parameters
- Apply control for graceful degradation
  
  ▶ Perform load shedding or degrade service under overload

Simplify task of building highly-concurrent services

- Decouple load management from service complexity
- Use of stages supports modularity, code reuse, debugging
- Dynamic control shields apps from complexity of resource management
Outline

• Problems with Threads and Event-Driven Concurrency
• The Staged Event-Driven Architecture
• SEDA Implementation
• Application Study: High-Performance HTTP Server
• Using Control for Overload Prevention
• Ongoing Work and Conclusions
Problems with Thread-Based Concurrency

- High resource usage, context switch overhead, contended locks
- Too many threads → throughput meltdown, response time explosion
- Traditional solution: Bound total number of threads
  - But, how do you determine the ideal number of threads?
- Regardless of performance, threads are fundamentally the wrong interface
  - Request stream hidden within scheduler
  - Transparency masks resource contention

(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)
Small number of event-processing threads with many FSMs

- Yields efficient and scalable concurrency
- Many examples: Click router, Flash web server, TP Monitors, etc.

Difficult to engineer, modularize, and tune

- Little OS and tool support: ‘‘roll your own’’
- No performance/failure isolation between FSMs
- FSM code can never block (but page faults, garbage collection force a block)
Decompose service into *stages* separated by *queues*

- Each stage performs a subset of request processing
- Stages internally event-driven, typically nonblocking
- Queues introduce execution boundary for isolation and conditioning

Each stage contains a *thread pool* to drive stage execution

- However, threads are not exposed to applications
- Dynamic control grows/shrinks thread pools with demand
  
  *Stages may block if necessary*

Best of both threads and events:

- Programmability of threads with explicit flow of events

Matt Welsh, UC Berkeley
Queues for Control and Composition

Queues subject to *admission control policy*
- e.g., Thresholding, rate control, credit-based flow control
  - *Applications must expect enqueue failures!*
- Block on full queue $\rightarrow$ backpressure
- Drop rejected events $\rightarrow$ load shedding
  - *May also take alternate action, e.g., degraded service*

Queues introduce explicit execution boundary
- Threads may only execute within a single stage
- Performance isolation, modularity, independent load management

Explicit event delivery supports inspection
- Trace flow of events through application
- Monitor queue lengths to detect bottleneck
Goal: **Determine ideal degree of concurrency for a stage**

- Dynamically adjust number of threads allocated to each stage
- Avoid wasting threads when unneeded

**Controller operation**

- Observes input queue length, adds threads if over threshold
- Idle threads removed from pool
Goal: *Schedule for low response time and high throughput*

- **Batching factor:** number of events consumed by each thread
- Large batching factor $\rightarrow$ more locality, higher throughput
- Small batching factor $\rightarrow$ lower response time

Attempt to find smallest batching factor with stable throughput

- Reduces batching factor when throughput high, increases when low
SEDA Prototype: Sandstorm

Implemented in Java with nonblocking I/O interfaces

- Scalable network performance up to 10,000 clients per node
- Influenced design of JDK 1.4 java.nio APIs

Java viable as service construction language

- Built-in threading, automatic memory management, cross-platform
  ▶ Java-based SEDA Web server outperforms Apache and Flash
Haboob: A SEDA-Based Web Server

Measured static file load from SpecWEB99 benchmark

- Realistic, industry-standard benchmark
- 1 to 1024 clients making repeated requests, think time 20ms
- Total fileset size is 3.31 GB; page sizes range from 102 Bytes to 940 KB

Maintains memory cache of recently accessed pages (200 MB)

- Significant fraction of page accesses require disk I/O

Comparison with Apache and Flash

- **Apache**: Process-based concurrency, 150 processes
  - Does not accept new TCP connections when all processes busy
- **Flash** (Vivek Pai, Princeton): Event-driven w/ 4 processes
  - Accepts only 506 simultaneous connections due to fd limits
Haboob Throughput vs. Apache and Flash

- **SEDA throughput 10% higher** than Apache and Flash (which are in C!)
  - *Some degradation due to Linux socket inefficiencies*
- Apache accepts only 150 clients at once - no overload despite thread model
  - *But as we will see, this penalizes many clients*

4-way Pentium III 500 MHz, Gigabit Ethernet, 2 GB RAM, Linux 2.2.14, IBM JDK 1.3
Response Time Distribution - 1024 Clients

<table>
<thead>
<tr>
<th></th>
<th>SEDA</th>
<th>Flash</th>
<th>Apache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean RT</td>
<td>547 ms</td>
<td>665 ms</td>
<td>475 ms</td>
</tr>
<tr>
<td>Max RT</td>
<td>3.8 sec</td>
<td>37 sec</td>
<td>1.7 minutes</td>
</tr>
</tbody>
</table>

- SEDA yields **predictable performance** - Apache and Flash are very unfair
  - "Unlucky" clients see long TCP retransmit backoff times
  - Everyone is "unlucky": multiple HTTP requests to load one page!
Dynamic Control for Overload Prevention

Arashi: Web-based e-mail service (Yahoo! Mail clone)

- Complex dynamic page generation, SSL encryption
- Mail stored in back-end MySQL database
- SEDA middle-tier conditions load on MySQL!

Adaptive admission control policy to meet performance target

- Dynamically adjust queue thresholds to maintain low response time
- Rejected clients sent friendly error message
  ▶ Could degrade service or redirect request instead
- Goal: 90th percentile response time of 1 sec
- Controller is ignorant of service logic

Performance with 128 clients:

<table>
<thead>
<tr>
<th></th>
<th>90th percentile RT</th>
<th>% requests rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>No control</td>
<td>7.5 sec</td>
<td>0%</td>
</tr>
<tr>
<td>With overload control</td>
<td>0.978 sec</td>
<td>49%</td>
</tr>
</tbody>
</table>
Ongoing Work

Formalize control-theoretic approach to resource management

- Large body of prior work in control of physical systems
- Internet services highly nonlinear, difficult to derive models
- Adaptive and fuzzy control as possible approaches

Generalize load conditioning mechanisms

- Extend resource control to memory, other resources
- General-purpose system overload monitor
- Explore degradation vs. load-shedding tradeoff

Ongoing implementation and application work

- Gnutella packet router
  - Peer-to-peer file sharing network
- Distributed, cluster-based SEDA (Berkeley Ninja Project)
  - Event queues implemented as network pipes
- Berkeley OceanStore Project using SEDA as a base
  - Global, secure file store and archival system
Summary

Support for massive concurrency requires new design techniques

- SEDA introduces service design as a network of stages
- Decouple load management from service complexity
- Expose request streams to applications for load conditioning

Observation and control as key to service design

- Dynamic control to keep stages within operating regime
- Controllers operate independent of application logic
- Bring body of work on control systems to bear on Internet services

Implications for OS and language design

- What would a “native” SEDA operating system look like?
- Language and tool support for event-driven computing
- SEDA opens up new questions in the service design space!

For more information, software, and (soon) my PhD thesis:

http://www.cs.berkeley.edu/~mdw/
Backup Slides Follow
Related Work

High-performance Web servers

- Many systems realizing the benefit of event-driven design
- \([\text{Flash, Harvest, Squid, JAWS, \ldots}]\)
- Specific applications - no general-purpose framework
- 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
How Complex is SEDA?

Code size and complexity

- Sandstorm runtime: 19934 LOC, 7871 NCSS
- 2566 NCSS for core runtime, 3023 NCSS for async I/O
- HTTP protocol library: 676 NCSS
- Haboob web server: 2607 NCSS

Some learning curve for event-driven programming

- Managing continuations, tracking events
- But, note that stages can block (for difficult code or lazy programmers)

Decomposition into stages helps greatly!

- Applications tend to map cleanly onto a pipeline of stages
- Each stage is a self-contained, well-conditioned module
- Typically little or no direct data sharing between stages
- Interposition of new stages is trivial

We have found SEDA to be much simpler and easier to reason about than other event-driven server frameworks