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

What does well-conditioned mean?

Internet service architectures
- thread per request
- thread pools
- event driven

The SEDA architecture

Evaluation
Internet Services

Wide variation in Internet load
  - the *slashdot* effect

Wide variation in service requirements
  - must support static and dynamic content
  - with responsiveness and high availability

Resource management challenge
  - supporting massive concurrency at a low cost
Well-Conditioned Services

A well-conditioned service will not bog down under heavy load!

As load increases response time should increase linearly

Well-conditioned services require the right architectural approach

SEDA = Staged Event-Driven Architecture
Architectural Alternatives

1. Thread per request architecture
2. Thread pool architecture
3. Event driven architecture
Thread Per Request

Create a new thread for each request
Delete thread when request is complete
Thread blocks during I/O
Standard approach in RPC, Java RMI, DCOM
Super Market Analogy

Hire a checkout clerk when a customer enters the store

Fire the checkout clerk when the customer leaves the store

Is this implementation of a super market service well-conditioned?

How could we do better?
Does This Work for Web Servers?
Why Does This Happen?

Despite being easy to program, this approach suffers from:

- excessive delay for thread creation
- overhead of thread destruction
- premature thrashing of CPU, memory, and cache, when load gets high
- high context switch overhead
- high memory costs for thread stack and TCB
Thread Pools

Very similar structure, except
- the number of threads is bounded
- threads are created statically
- threads are recycled after use
- requests delayed when all threads in use

Standard approach in Apache, IIS, Netscape
ES, BEA Weblogic, IBM WebSphere, etc
Super Market Analogy

Hire N permanent checkout clerks
- each customer is assigned to a clerk
- M clerks per cash register
- clerks may need to queue to use register

How does this approach perform during normal load and during overload?

What happens if a customer has an unusual request?
Thread Pool Performance

Mixed workloads can result in unfair delays

It's difficult to identify problems or sources of bottlenecks when all threads look alike

It's difficult to know how big the thread pool should be
The Event-Driven Approach

Request arrival is an event

Events are handled by the execution of a function
- an event handler

Event handlers are run sequentially and non-preemptively
- using one thread per CPU
Super Market Analogy

One checkout clerk per cash register

Customers queue waiting for clerk

Clerk completes work for one customer before starting work for the next

Bottlenecks are easy to identify and fix
- customer queues get too long at the problem register
- customers can be moved from one queue to another
Does This Work for a Web Server?
Event Driven Architectures

What is good about them?
- Robust in the face of load variation
- High throughput
- Potential for fine grain control
Event Driven Architectures

Used in Flash, thttpd, Zeus, JAW, and Harvest

SEDA extends the idea to expose load-related information and to simplify and automate dynamic load balancing and load shedding
SEDA’s Building Block – Stage
Stages Connected by Event Queues

Event queues define the control boundaries and can be inspected!
Dynamic Resource Controllers

(a) Thread pool controller
(b) Batching controller
Thread Pool Controller Performance

- PageCache
  - check cache
- CacheMiss
  - handle miss
  - I/O request
  - file I/O
- Event Handler
  - Thread Pool
    - Observe Length
    - Adjust Size
    - Threshold

Graphs showing performance metrics for different client counts and time intervals.
Batching Controller Performance

Diagram showing the process flow of file data handling, cache miss handling, and file I/O. The diagram includes a flowchart with steps such as 'PageCache', 'check cache', 'CacheMiss', 'handle miss', and 'file I/O'. There is also a graph showing the output rate (events/sec) over time (100 ms intervals) with a running average and batching factor.
Adaptive Load Shedding
Asynchronous I/O

You need asynchronous I/O but it’s not always available from the OS!

Asynchronous socket I/O
- non-blocking socket calls used in readStage, writeStage, and listenStage

Asynchronous file I/O
- asynchronous file calls not available
- had to fake it using a thread pool
Haboob HTTP Server

(a) Throughput vs. number of clients

(b) Cumulative distribution of response time for 1024 clients

<table>
<thead>
<tr>
<th>Server</th>
<th>Throughput</th>
<th>256 clients</th>
<th>1024 clients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RT mean</td>
<td>RT max</td>
</tr>
<tr>
<td>Apache</td>
<td>173.36 Mbps</td>
<td>143.91 ms</td>
<td>27953 ms</td>
</tr>
<tr>
<td>Flash</td>
<td>180.83 Mbps</td>
<td>141.39 ms</td>
<td>10803 ms</td>
</tr>
<tr>
<td>Haboob</td>
<td>208.09 Mbps</td>
<td>112.44 ms</td>
<td>1220 ms</td>
</tr>
</tbody>
</table>
Gnutella Packet Router

(a) Using single thread

(b) Using thread pool controller

(c) Queue length profile
Conclusion

The SEDA approach works well

- it supports high concurrency
- it is easy to program and tune
- services are well-conditioned
- introspection and self-tuning are supported