SEDA
An architecture for Well-Conditioned, scalable Internet Services

Matt Welsh, David Culler, and Eric Brewer
University of California, Berkeley
Symposium on Operating Systems Principles (SOSP), October 2001

http://www.eecs.harvard.edu/~mdw/proj/seda/

Presented by:
Disha Gandhi
Outline

• Why SEDA ??
• What is SEDA ??
• Thread and Event-based Concurrency
• SEDA Architecture
• Asynchronous I/O Primitives
• Applications and Performance Evaluation
Motivation for SEDA

• Prototypical application: web server
• Requirements for internet services
  – Need high level concurrency.
  – Handle huge variations in load (Slashdot effect)
  – Handle new system designs to manage these loads challenges – dynamic content, hosting of services on general-purpose servers.
• Well-conditioned service: simple pipeline without degradation beyond load saturation
What is SEDA???

• From Matt Welsh’s PhD work at Berkeley (2000-2002)
• A design for highly concurrent server applications
• Combines elements of both thread and event-based programming
• Dynamic resource control mechanisms for automatic tuning and load conditioning to handle large fluctuations in load
Thread based concurrency model

- Thread per-request model (commonly used design for server applications)
- Synchronization protects shared resources
- OS overlaps computation and I/O by transparently switching among threads.
- Cache/TLB misses, threads scheduling, lock contention overheads.
Thread-server performance degradation

- High load - high context overhead.
- Throughput decreases
- Response time very high as task queue length increases
- Not good for concurrency requirements of Internet
Bounded Thread Pools

• To avoid overuse of threads, bind the size of thread pool.
• Reject additional connections.
• Common solution (Apache etc)
• Throughput, overall performance better

But :
• is it fair to clients?
• How to determine upper limit?
• Difficult to identify internal performance bottlenecks.
Pipeline Model Of Computation

Fig: Pipeline Model of Computation
Event-driven concurrency

- Small number of threads (typically one per CPU) that loop continuously, processing events from a queue.
- Events-handlers should be short-lived.
- I/O must be non-blocking.
Performance

- Robust to load
- Throughput constant
- Latency linear.
Problems with Event-driven

• Scheduling and overhead of events
• Choice of scheduling algorithm is tailored to application.
• No principled framework, so solutions are typically ad-hoc, lacking modularity
• “Structured Event queues” to improve code modularity and simplify application design.
Staged Event-Driven Architecture

- Application decomposed into independent stages separated by queues.
- Supports massive concurrency: each stage has own thread pool
- Self-contained application components—event handler, incoming event queue and thread pool—clean boundary for modularity/security and introspection. (at cost of increased latency)

- Finite event queues can provide:
  - backpressure: blocking on a full queue
  - load shedding: dropping events
  - errors to user, etc.
Anatomy of a Stage

- Self-contained application components.
- Application-supplied event handler
- Controller dynamically adjusts resource allocation.
- Small number of threads per stage
Dynamic Resource Controller

- Automatically adapt the resource usage of a stage
- based on observed performance and demand

Adjust number of threads allocated to stage based on actual measured usage

- Adjust batching factor to increase throughput while maintaining cache optimization and task aggregation
Using Asynchronous I/O Primitives

Asynchronous socket I/O
- Uses non-blocking I/O provided by OS for services.

Asynchronous file I/O
- Uses blocking I/O and a bounded threaded pool.
Comparison

- SEDA based layer: non-blocking I/O by OS and /dev/poll event-driven mechanism: $\sim$constant I/O bandwidth
- Compatibility layer that uses blocking sockets and a thread pool: thread thrashing
Sandstorm: A SEDA Prototype

- Entirely in java
- Native libraries for non-blocking socket I/O
- Automated memory management to garbage-collecting expired events.
- Each application module implements a simple event handler interface with a single method call which processes a batch of events pulled from stages’s incoming event queue.
- Run-time system and associated controllers create threads.
- API’s for naming, creating and destroying stages, and other functions.
- Socket and file I/O mechanisms as standard interfaces.
Haboob : SEDA Web server

- More complicated web server: 10 stages, 4 for asynchronous socket and disk I/O.
- HttpParse stage: accept new client connections.
- HttpRecv stage: accept HTTP connection and request events and pass to PageCache stage or generate response directly
- PageCache: in-memory Web page cache using hashtable indexed by URL
- Cachemiss: handle page cache misses, using asynchronous file I/O to read from disk
- HttpSend: send response to client, statistics gathering
- High modularity
Performance Comparison

Throughput stable at high loads for all, though Haboob is the best.
Apache and Flash have huge variation in response time (long tails)
Haboob has low variation in response time, but has longer average response time
Haboob has graceful degradation, apache fairness decline quickly after its limit of number of connections.
Results: Resource Throttling

- 1024 clients repeatedly request dynamic pages with I/O and computation
- Apache and Haboob (with no control) process all requests, Flash rejects some due to a bug in its code.
- Haboob controller drops (with error) if average response time > 5 sec
Gnutella packet router

- Use of SEDA for non-traditional Internet services.
- Routing packets in a peer to peer file sharing network
- 3 stages in SEDA implementation: GnutellaServer, GneutellaRouter and Gneutellacatcher
Summary

• SEDA provides a framework for web services (handle massive concurrency and huge variations in load).
• Combined approach tries to achieve performance, with flexibility and ease of programming.
• SEDA uses dynamic controllers for resource usage and scheduling at each stage.
• Robust performance as per published results.
Current state of SEDA

• “a number of recent research papers have demonstrated that the SEDA prototype (in Java) performs poorly compared to threaded or event-based systems implemented in C”

• “The most fundamental aspect of the SEDA architecture is the programming model that supports stage-level backpressure and load management. Our goal was never to show that SEDA outperforms other server designs, but rather that acceptable performance can be achieved while providing a disciplined approach to overload management. Clearly more work is needed to show that this goal can be met in a range of application and server environments.”

• http://www.eecs.harvard.edu/~mdw/proj/seda/
Acknowledgement

- Previous class presentations.