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

Link to paper

Abstract

- Staged event-driven architecture (SEDA) consists of writing services that are composed of individual, scalable stages connected by explicit queues.
- Makes use of dynamic resource controllers in order to scale stages with load (thread pool sizing, event batching, adaptive load shedding)

Background

- SEDA combines two things:
    a. thread-based concurrency models
    b. event-based models
- A well-conditioned service exhibits graceful degradation - queuing delay should dominate client latency (that increases linearly with number of clients), and actual throughput should NOT degrade.
- It should also behave like a simple pipeline, with performance determined simply by the depth of the pipe under minimal load

Thread-based Concurrency

- Most common model - thread per request (RPC, Java Remote Method Invocation, DCOM), rely on thread sychronization primitives to coordinate
- Threads have issues - cache / TLB misses, lock contention, thread allocation
- Virtualization also muddies the waters - modern OSes provide safe, containerized access to resources, which means that applications and schedulers have very little input into synchronization

Bounded thread pools

- Bounded thread pools let a service offer reliable performance at the cost of denied / delayed service to clients after thread saturation. Big dangerous trade-off...
- Different threads / requests traditionally correspond to different resource demands and contention - fixed thread pooling doesn't allow a service to introspect once threads are saturated (although this seems easy to circumvent..)

Event-Driven concurrency

- Concept of FSMs (finite state machines) and how they handle continuations without threads is really interesting, although I don't follow how the memory footprint isn't comparable (and is less? are they cached? that'd be cool)
- Each task is implemented as a FSM that responds to events
- Each CPU usually has one thread dedicated to processing a certain kind of event and updating FSMs accordingly
- This does rely heavily on a scheduler to coordinate the evolution of the FSMs
- Ordering comes in as a big concern too - in short, you need application-specific, robust schedulers to multiplex event streams to the proper event-handling threads
- This model assumes event-handling threads are non-blocking - synchronous operations like IO should be dispatched to a blocking call elsewhere and register a callback upon completion (interrupt model?)

The Staged Event-Driven Architecture

- Goals are:
    - Event-driven to help support massive concurrency
    - Modular construction, profiling and debugging, as well as abstractions to avoid underlying scheduling / queuing mechanisms are all crucial
    - Introspection - inspect current workload to prioritize and filter
    - Self-tuning resource allocation - stages should be able to adjust primitives like thread pools in response to contention

Stages as robust building blocks

- Stage is the atomic unit - it's an independent application consisting of:
    - Event handler
    - Incoming event queue
    - Thread pool
- Pull a batch off the queue via the event handler, then dispatch / multiplex zero or more events to separate stages based on results - this is the whole shebang
- Number of threads is small per stage, and can be elastically responsive to demand.
- Stages can run concurrently or serially - this white paper assumes preemptive OS threads in an SMP context

Applications as networks of stages

- Stages are distinct and modular, and queues enforce this separation
- Queues can enforce backpressure (blocking on a full queue) or work shedding (dropping work on a full queue)

Dynamic resource controllers

- Resource controllers observe performance and adjust resource usage accordingly
- Performance can be observed both entirely locally to a stage OR globally in an application

- Two resource controllers - thread pool controller, and batching controller
    - The thread pool controller periodically samples the input queue and allocates threads based on some preconfigured rate of contention
    - Batching controller - don't want to increase point latency, but batching increases cache locality, TLB hits, better memory access patterns, all the usual stuff. Maintains a moving average of output events and increases the batch size until throughput begins to degrade beyond some threshold
- SEDA's controllers are agnostic to OS resources and implementations - they're application-level mechanisms

There's more to this paper, two more sections, but I'm not terribly interested in a bunch of implementation work, mostly theory.