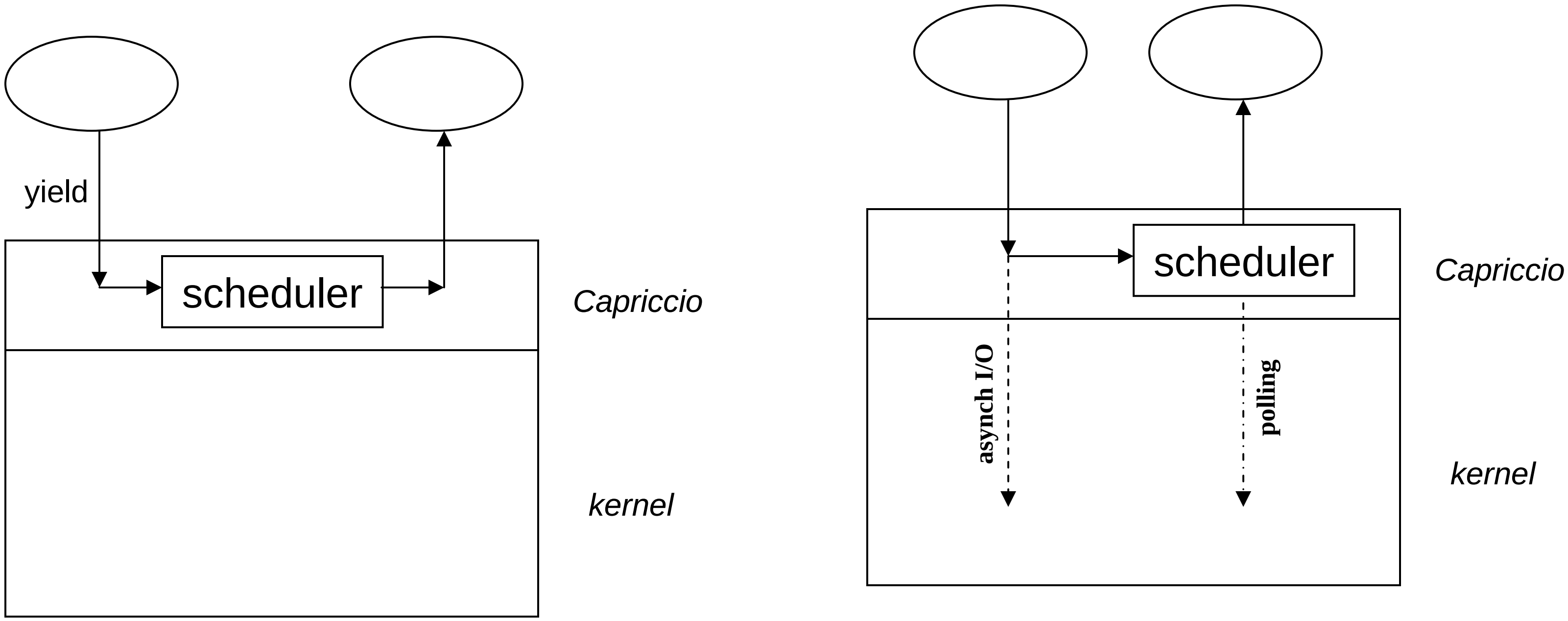
Threads vs. Events

SEDA – An Event Model

Cappricio

- Philosophy
  - Thread *model* is useful
  - Improve *implementation* to remove barriers to scalability
- Techniques
  - User-level threads
  - Linked stack management
  - Resource aware scheduling
- Tools
  - Compiler-analysis
  - Run-time monitoring

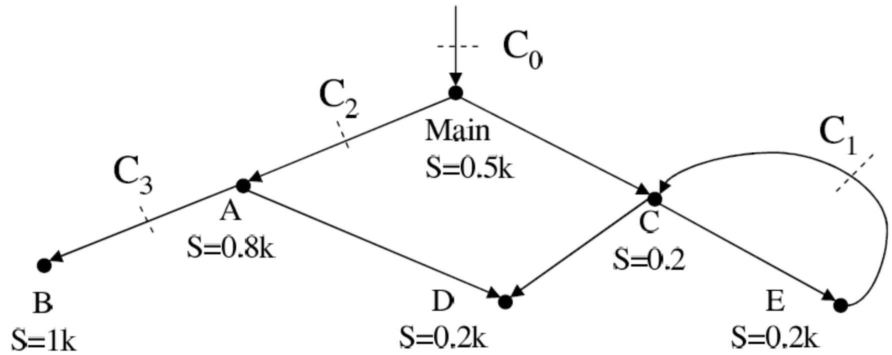
Capriccio – user level threads



- User-level threading with fast context switch
- Cooperative scheduling (via yielding)
- Thread management costs independent of number of threads (except for sleep queue)

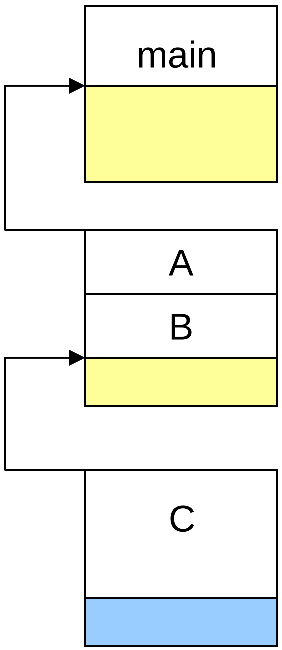
- Intercepts and converts blocking I/O into asynchronous I/O
- Does polling to determine I/O completion

Compiler Analysis - Checkpoints



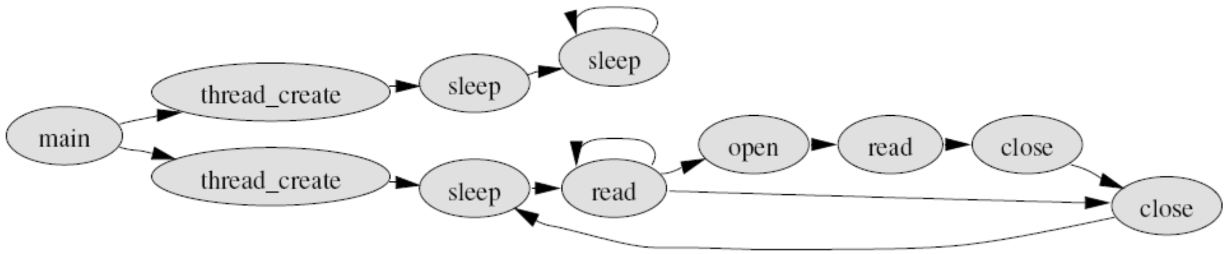
- Call graph – each node is a procedure annotated with maximum stack size needed to execute that procedure; each edge represents a call
- Maximum stack size for thread executing call graph cannot be determined statically
  - Recursion (cycles in graph)
  - Sub-optimal allocation (different paths may require substantially different stack sizes)
- Insert checkpoints to allocate additional stack space ("chunk") dynamically
  - On entry (e.g., C_O)
  - On each back-edge (e.g. C_1)
  - On each edge where the needed (maximum) stack space to reach a leaf node or the next checkpoints exceeds a given limit (*MaxPath*) (e.g., C_2 and C_3 if limit is 1KB)
- Checkpoint code added by source-source translation

Linked Stacks



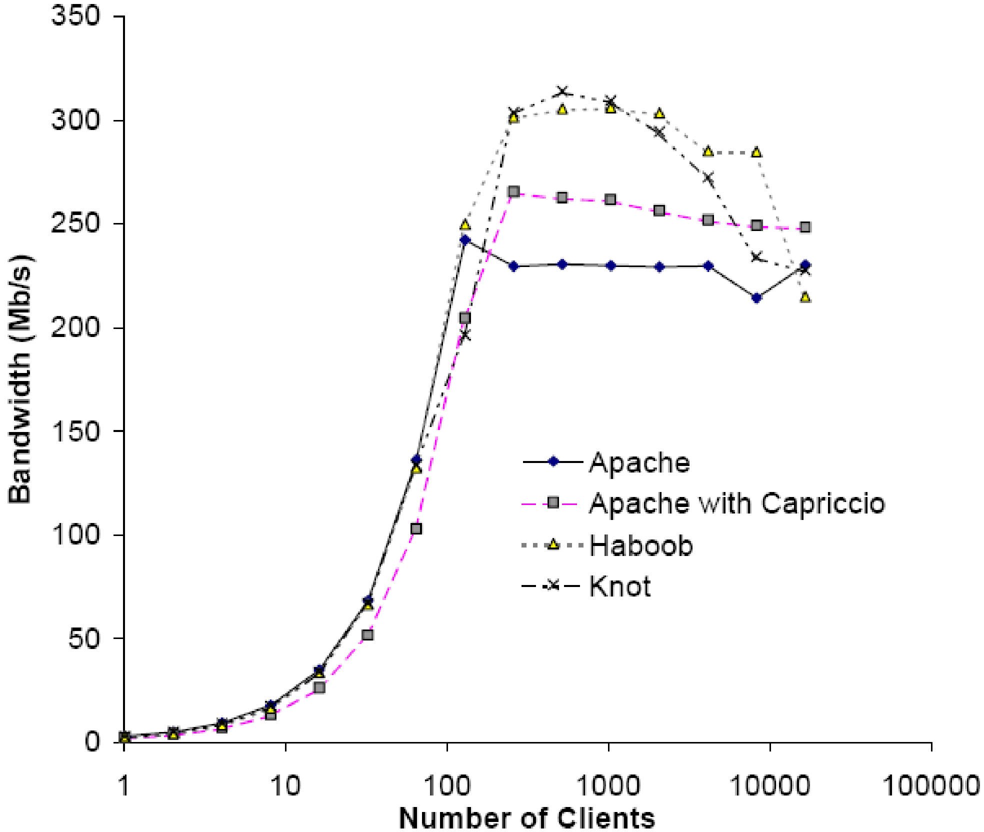
- Thread stack is collection of non-contiguous blocks ('chunks")
- *MinChunk:* smallest stack block allocated
- Stack blocks "linked" by saving stack pointer for "old" block in field of "new" block; frame pointer remains unchanged
- Two kinds of wasted memory
  - Internal (within a block) (yellow)
  - External (in last block) (blue)
- Two controlling parameters
  - MaxPath: tradeoff between amount of instrumentation and run-time overhead vs. internal memory waste
  - MinChunk: tradeoff between internal memory waste and external memory waste
- Memory advantages
  - Avoids pre-allocation of large stacks
  - Improves paging behavior by (1) leveraging LIFO stack usage pattern to share chunks among threads and (2) placing multiple chunks on the same page

Resource-aware scheduling



- Blocking graph
  - Nodes are points where the program blocks
  - Arcs connect successive blocking points
- Blocking graph formed dynamically
  - Appropriate for long-running program (e.g. web servers)
- Scheduling annotations
  - Edge – exponentially weighted average resource usage
  - Node – weighted average of its edge values (average resource usage of next edge)
  - Resources – CPU, memory, stack, sockets
- Resource-aware scheduling:
  - Dynamically prioritize nodes/threads based on whether the thread will increase or decrease its use of each resource
  - When a resource is scarce, schedule threads that release that resource
- Limitations
  - Difficult to determine the maximum capacity of a resource
  - Application-managed resources cannot be seen
  - Applications that do not yield

Performance comparison



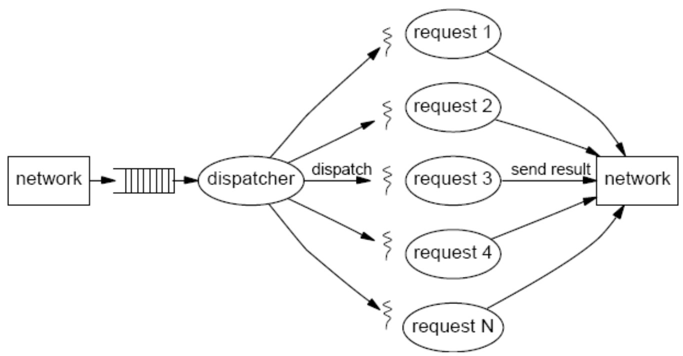
- Apache – standard distribution
- Haboob – event-based web server
- Knot – simple, threaded specially developed web server

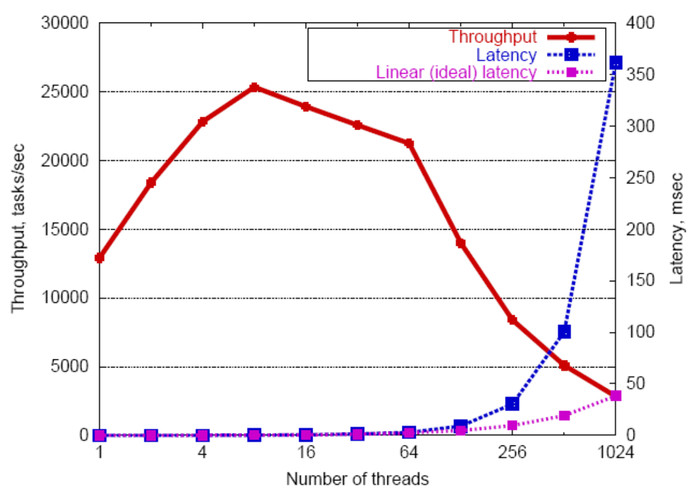
SEDA – Staged Event-Driven Architecture

- Goals
  - Massive concurrency
    - required for heavily used web servers
    - large spikes in load (100x increase in demand)
    - requires efficient, non-blocking I/O
  - Simplify constructing well-conditioned services
    - "well conditioned": behaves like a simple pipeline
    - offers graceful degradation, maintaining high throughput as load exceeds capacity
    - provides modular architecture (defining and interconnecting "stages")
    - hides resource management details
  - Introspection
    - ability to analyze and adapt to the request stream
  - Self-tuning resource management
    - thread pool sizing
    - dynamic event scheduling

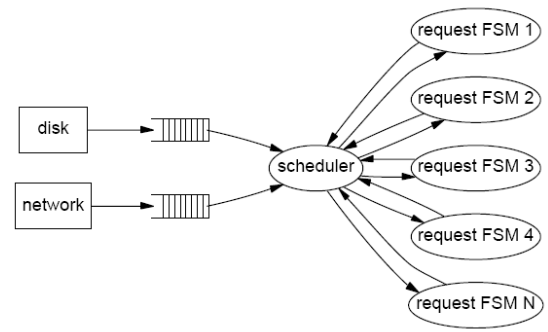
- Hybrid model
  - combines threads (within stages) and events (between stages)

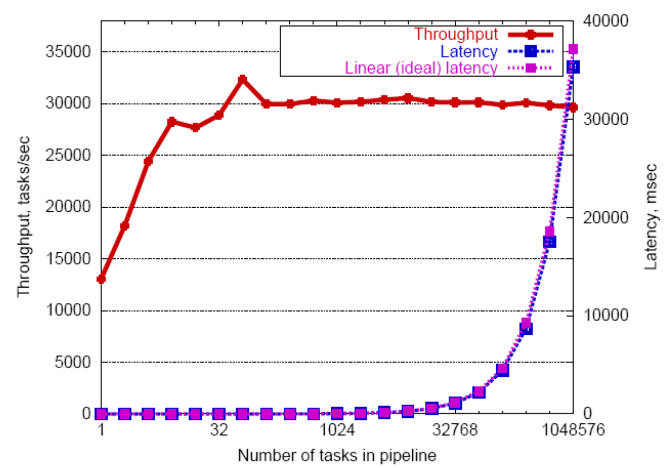
SEDA's point of view





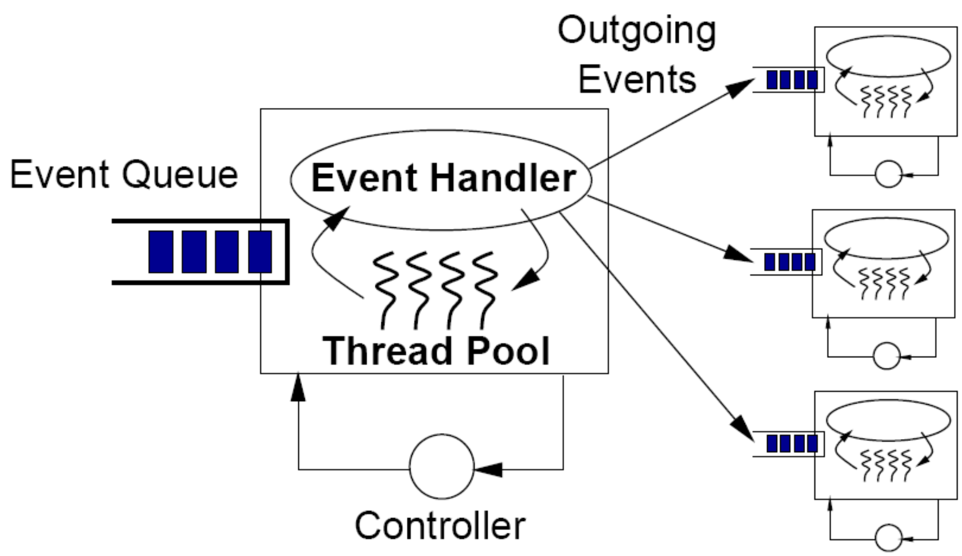
Thread model and performance





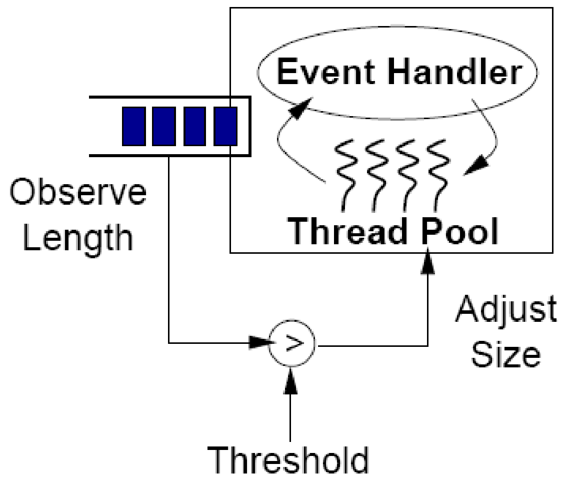
Event model and performance

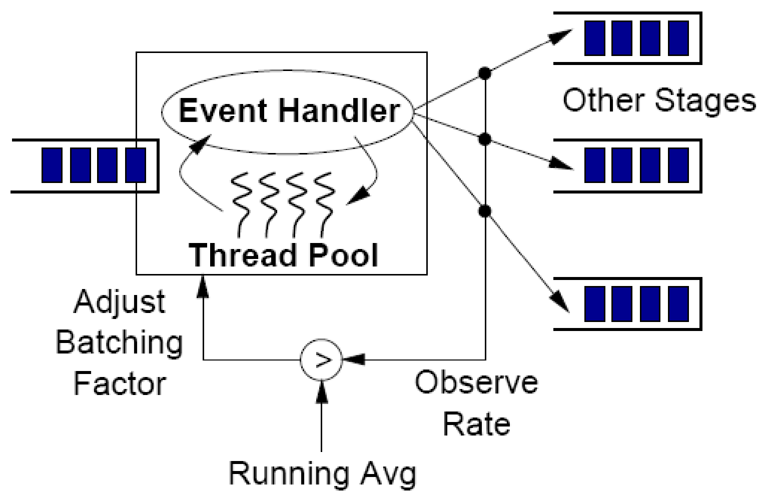
SEDA - structure



- Event queue – holds incoming requests
- Thread pool
  - takes requests from event queue and invokes event handler
  - Limited number of threads per stage
- Event handler
  - Application defined
  - Performs application processing and possibly generates events for other stages
  - Does not manage thread pool or event queue
- Controller – performs scheduling and thread management

Resource Controllers





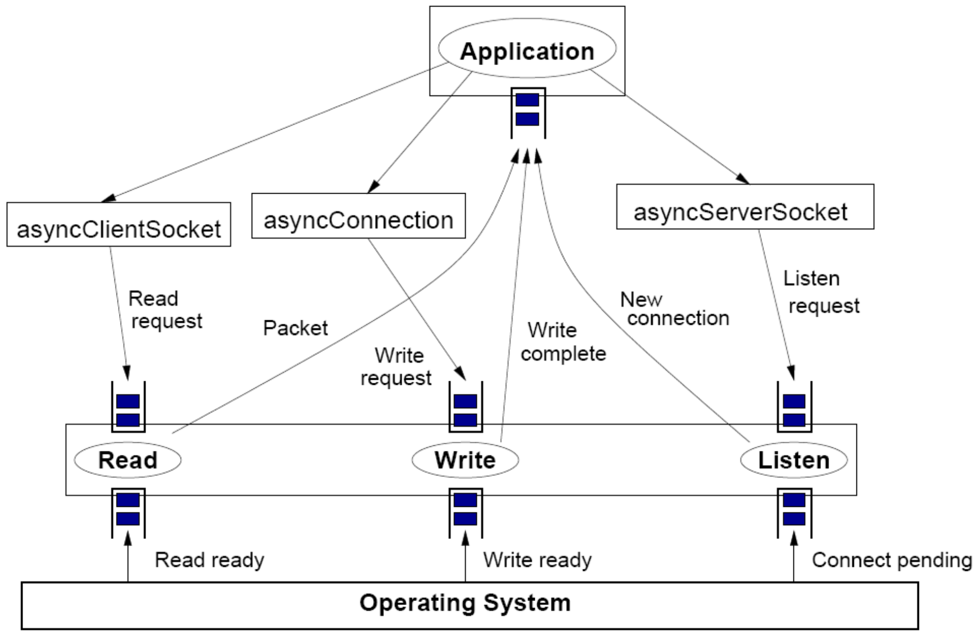
Thread pool controller

- Thread added (up to a maximum) when event queue exceeds threshold
- Thread deleted when idle for a given period

Batching controller

- Adjusts batching factor: the number of event processed at a time
- High batching factor improves throughput
- Low batching factor improves response time
- Goal: find lowest batching factor that sustains high throughput

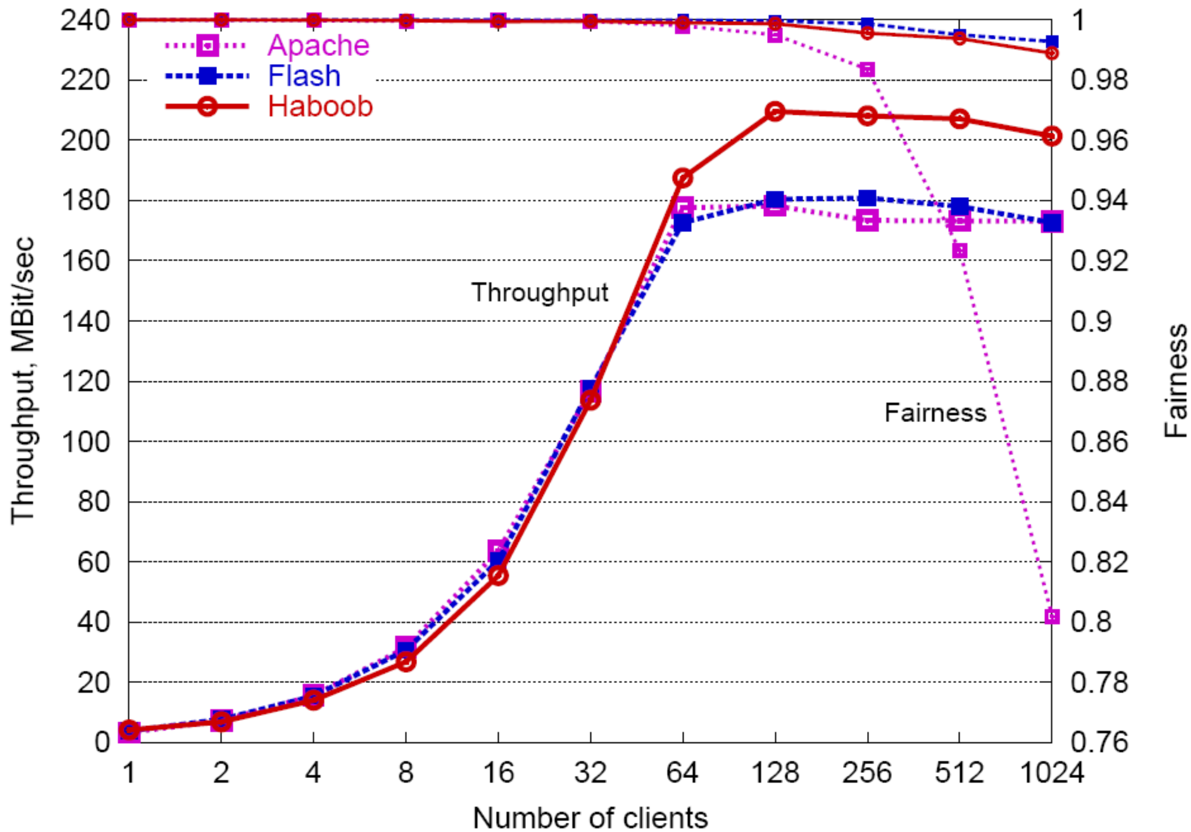
Asynchronous Socket layer



- Implemented as a set of SEDA stages
- Each asynchSocket stage has two event queues
- Thread in each stage serves each queue alternately based on time-out
- Similar use of stages for file I/O

Performance

- Apache
  - process-per-request design
- Flash
  - event-drived design
  - one process handling most tasks
- Haboob
  - SEDA-based design



Fairness

- Measure of number of requests completed per client
- Value of 1 indicates equal treatment of clients
- Value of k/N indicates k clients received equal treatment and n-k clients received no service