JBoss Seam Performance and Scalability on Dell PowerEdge 1855 Blade Servers

Dave Jaffe, PhD, Dell Inc.
Michael Yuan, PhD, JBoss / RedHat

June 14th, 2006

About us

- Dave Jaffe
  - ✓ Works for Dell Scalable Enterprise Technology Center
  - ✓ Author of 30+ articles including 20 Dell Power Solutions articles
  - ✓ Creator of the DVD Store test application
- Michael Yuan
  - ✓ Works for JBoss
  - ✓ Author of 5 books and 50+ articles

Agenda

- The basics
- The test tools and environment
- Interpret performance tests results
- Optimize Seam on a single server
- Load balanced cluster
- In the real world

Testing done in June 2006

What is Seam?

- An annotation framework to tie together EJB3, JSF, jBPM, AJAX etc.
- Advanced state management facilities
  - ✓ Finely grained stateful contexts beyond HTTP session
  - ✓ Multiple concurrent browser windows / tabs support
  - ✓ Long running business processes with multiple users
- POJO-based lightweight framework

Visit the JBoss booth and attend tomorrow's Seam lab for more!

Productivity and Performance

Seam makes it very easy for developers to write advanced web applications with powerful state management facilities, business process integration, AJAX UI …

However, is there a trade-off for the vastly improved developer productivity? Can Seam applications handle as high a load as other Java EE applications?

Seam performance implications

- Pros
  - ✓ Extended persistence context could be much faster than database transactions
  - ✓ Static annotation processing is often faster than XML parsing
  - ✓ Eliminate HTTP session-based memory leaks
- Cons
  - ✓ Runtime annotation processing
  - ✓ Runtime dependency bi-jection
  - ✓ Multiple interceptors

The cons are mostly for runtime business logic

We need to answer …

- Does Seam business logic components perform well enough for most enterprise applications on state of art hardware?
- How to tune Seam web applications?
- Do Seam web applications scale in a cluster environment?

Proven use cases at low load: We run EJB3/Seam applications on our own production servers, including demo.jboss.com, jboss.org, and our internal support portal. Those servers handle at least several thousand unique visitors every day.

7

Agenda

- The basics
- The test tools and environment
- Interpret performance tests results
- Optimize Seam on a single server
- Load balanced cluster
- In the real world

Testing done in June 2006

8

Dell Scalable Enterprise Technology Center Lab

- 10 PowerEdge 1855 Blade Servers
- Gigabit Ethernet
- KVM switch for ease of management

9

Dell PowerEdge 1855 Blade Server

- Dual Xeon EM64T 64 bit processors
- Each is dual core
- Hyper-threading available (8 logical CPUs in one blade)
- 2.8 GHz
- 8GB RAM
- 73 GB mirrored disks
- 10 blades in a 7U rack
- Easy wiring
- Low power consumption

10

Software stack

- SUSE Linux Enterprise Server 9 SP3 for X86_64
- 64-bit JDK 5.0 from SUN
- Servers
  - ✓ JBoss AS 4.0.4 GA
  - ✓ Seam 1.0.0 RC3
  - ✓ Apache 2.0 with Mod_JK 1.2.15 load balancer
- Clients
  - ✓ Grinder 3.0
  - ✓ Home-brew C# driver for sanity check

11

Grinder 3.0

- Fully featured web stress testing suite
- Completely open source
- Fully programmable testing scripts in Jython
- 30,000 testing threads on a single Dell PE1855 Blade
- Ramp up the load / threads in any way you want
- Rich statistics reporting
- Testing agents on multiple machines
- Visual console for consolidated statistics

12

Jython script

- Supports all common HTTP features (proxy, authentication, cookies etc.)
- Make any request (or group of requests) a test and gather statistics on it
- Constructs new requests programmatically based on HTML returned from previous requests
- Can be auto-recorded from a browser session
- Arbitrary or random “think time” between requests
- Detailed logging and statistics reporting

13

An example script recorded by Grinder

```
def page2(self):
  """POST search.seam (request 201)."""
  result = request201.POST('/shopping-seam/search.seam',
    ( NVPair('jsf_tree_64', self.token_jsf_tree_64),
      NVPair('jsf_state_64', self.token_jsf_state_64),
      NVPair('jsf_viewid', self.token_jsf_viewid),
      NVPair('_id0:_id1', productSize),
      NVPair('_id0:_id2', 'Search'),
      NVPair('_id0_SUBMIT', self.token__id0_SUBMIT),
      NVPair('_id0:_link_hidden_', ''), ),
    ( NVPair('Content-Type', 'application/x-www-form-urlencoded'), ))
  self.token_jsf_tree_64 = \
    httpUtilities.valueFromHiddenInput('jsf_tree_64') #
'H4sIAAAAAAAAAK2Sv0oDQRDGx6iIIohoWAgiKKgg...'
  self.token_jsf_state_64 = \
    httpUtilities.valueFromHiddenInput('jsf_state_64') #
'H4sIAAAAAAAALVVzW8bRRQfrS0mbVMpTlAkaAqp...'
  self.token_jsf_viewid = \
    httpUtilities.valueFromHiddenInput('jsf_viewid') # '/product.jsp'

  return result
```

14

Customize the script

```
grinder.sleep(rand.nextInt(5000))
self.page2()      # POST search.seam (request 201)

grinder.sleep(rand.nextInt(5000))
self.page3()      # POST product.seam (request 301)

grinder.sleep(rand.nextInt(5000))
self.page4()      # POST product.seam (request 401)

grinder.sleep(rand.nextInt(5000))
self.page5()      # POST product.seam (request 501)
```

15

Test application

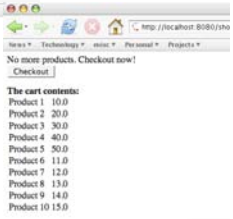
- Mimics a shopping cart
- Web tier only -- no database
- Simple workflow
  - ✓ Start a shopping cart
  - ✓ Add 10 products one by one into the cart (about 10KB data by default)
  - ✓ Eliminate the shopping cart content and start over again

16

The test application in action

1. Start the session. The product “size” is used to control the size of the stateful cart.

2. Add 10 products to the cart.



3. Clear all contents from the cart and start again.

17

Agenda

- The basics
- The test tools and environment
- Interpret performance tests results
- Optimize Seam on a single server
- Load balanced cluster
- In the real world

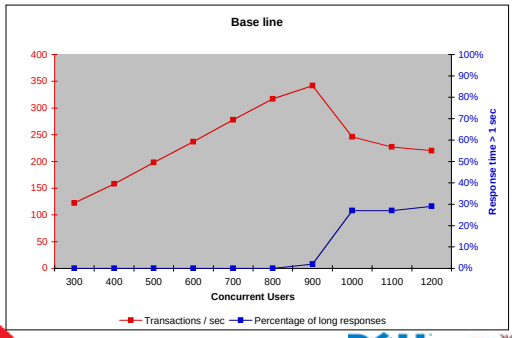
18

Control variables and measurements

- Concurrent # of users (threads in the driver program)
- Average think time
- Web Transactions per second (TPS)
- Response time distribution
- CPU and resource utilization

19

Stress testing curve



20

Stage 1: The server is under-utilized

- Server resources (e.g., CPU, memory, I/O) are utilized below 85% capability
- Server can throw in more resources to handle more requests
- The response time is very small (<50ms)
- TPS scales up with the load the users can generate
  - ✓ TPS ~ (Number of Threads) / (Average think time)

21

Stage 2: The server is fully loaded

- Server resources usage reaches 85% to 99%
- Sustainable TPS reaches a peak
- This is the place to be for most data center operators
- After that, the server must increase response time to slow down the users
  - ✓ TPS ~ (Number of Threads) / (Average think time + average response time)
  - ✓ The response time must rapidly increase to the level of think time

22

Stage 3: The server halts

- Server resources are utilized at constant 99%
- The response time become untolerable to users and the application “freezes”
  - ✓ Need long response time comparable to “think time” to slow down the users
  - ✓ With occasional GC, some requests could take more than 10s to respond
  - ✓ As the server becomes congested, it handles less transactions / sec and hence requires even longer response time
- Lots of connection timeouts and web server (500) errors

23

Think time vs. response time is crucial

- Think time >> normal response time is needed to reach the peak TPS
  - ✓ Otherwise, the response time starts to slow down the driver long before the peak
  - ✓ Short think time causes built-in relationship between load, TPS and response time, which interferes with the relationship we want to study
- Think time gives us more control over how to ramp up the load
- We choose a random think time between 0s to 5s
- Think time helps us scale test results to real world scenarios

24

From test to real users

- Real users may require much more think time than 5s
- In theory, if the real user needs up to 50s think time, a real world server can handle 10x concurrent users than the test threads
- In reality, the scale from test threads to real users might not be linear with think time:
  - ✓ More users require more resources (e.g., HTTP sessions, sockets, and thread switching)

25

Agenda

- The basics
- The test tools and environment
- Interpret performance tests results
- Optimize Seam on a single server
- Load balanced cluster
- In the real world

26

JVM options

- Allocate a lot of RAM for the JVM
  - ✓ We allocate 6GB on each of our blades (we use the 64bit JVM here!)
  - ✓ Memory is not a bottleneck in all our tests
  - ✓ Edit the bin/run.conf file and use JVM startup option: -Xmx 6g -Xms 6g
- Must use the -server option (default)
- Choose the parallel GC strategy to avoid long pause
  - ✓ -XX:+UseParallelGC -XX:+UserParallelOldGC

27

Problems with large RAM

- The GC takes much longer to clean up a large heap.
  - ✓ Seam generates a lot of objects for the GC since it does not cache in HTTP session (and hence avoid memory leak)
- Solutions
  - ✓ Run multiple JVMs on the same blade
    - Bind JBoss instances to specific port ranges
    - Bind JBoss instances to different IP
  - ✓ Run multiple virtual machines on the same blade (each VM has its own IP address)
  - ✓ External load balancer is needed

28

Turn off excessive logging

- Seam logs a lot of debug info by default
- Extensive disk I/O degrades server performance at high load
- Increase the logging level for the org.jboss packages to INFO in server/default/conf/log4j.xml

29

Client vs Server side state saving

- Client state saving uses more CPU / network
  - ✓ Serialization of state objects is slow
  - ✓ About 20kB of extra data in each page
- Server state saving uses more RAM
  - ✓ More difficult to cluster
  - ✓ Our servers are not limited by RAM

30

Configure state saving method

In web.xml file:

```
<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
</context-param>
```

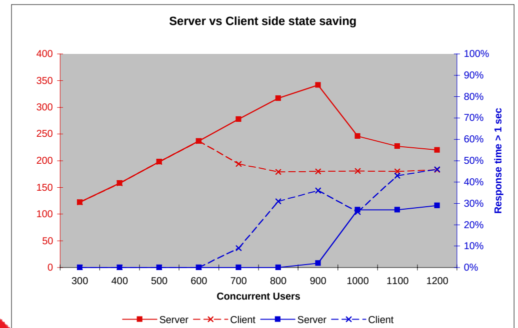
Server side state saving:

```
<form id="_id0" name="_id0" method="post"
action="/shopping-seam/checkout.seam"
enctype="application/x-www-form-urlencoded">
No more products. Checkout now!<br/>
<input id="_id0:_id1" name="_id0:_id1"
type="submit" value="Checkout"
onclick="clear__5Fid0();" />
<input type="hidden" name="_id0_SUBMIT"
value="1" /><input type="hidden"
name="_id0:_link_hidden_" /><script
type="text/javascript"><!--
function clear__5Fid0() {
  var f = document.forms['_id0'];
  f.elements['_id0:_link_hidden_'].value='';
  f.target='';
}
clear__5Fid0();
//--></script></form>
```

Client side state saving:

```
<form id="_id0" name="_id0" method="post"
action="/shopping-seam/checkout.seam"
enctype="application/x-www-form-urlencoded"><input type="hidden"
name="jsf_tree_64" id="jsf_tree_64"
value="[illegible]"
/><input type="hidden" name="jsf_state_64" id="jsf_state_64"
value="[illegible]"
/><input type="hidden" name="jsf_viewid" id="jsf_viewid"
value="/checkout.jsp" />
No more products. Checkout now!<br/>
<input id="_id0:_id1" name="_id0:_id1" type="submit"
value="Checkout" onclick="clear__5Fid0();" />
<input type="hidden" name="_id0_SUBMIT" value="1" /><input
type="hidden" name="_id0:_link_hidden_" /><script
type="text/javascript"><!--
function clear__5Fid0() {
  var f = document.forms['_id0'];
  f.elements['_id0:_link_hidden_'].value='';
  f.target='';
}
clear__5Fid0();
//--></script></form>
```

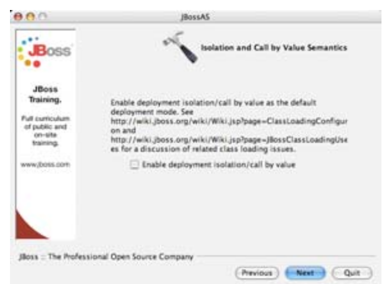

Tests Results



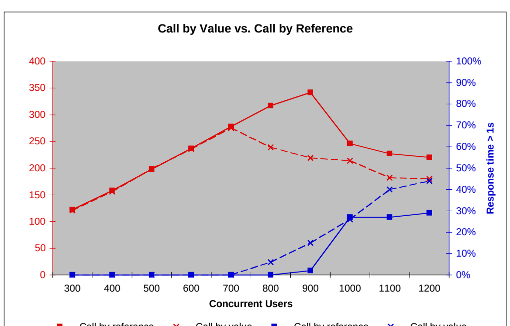
Call-by-value vs call-by-ref

- Seam makes a lot of calls from the web module (war) to the business module (ejb3 jar) via dynamic proxies
- Call-by-value
  - ✓ Important if you have multiple versions of the same class on the same server
  - ✓ This is the Java EE spec-complaint behavior
  - ✓ Requires CPU intensive serialization
- Call-by-reference
  - ✓ Designed to be faster than call-by-value for each call
  - ✓ Can give you odd problems if you port another Java EE app to JBoss
  - ✓ Not a problem for most Seam apps

Specify call semantics



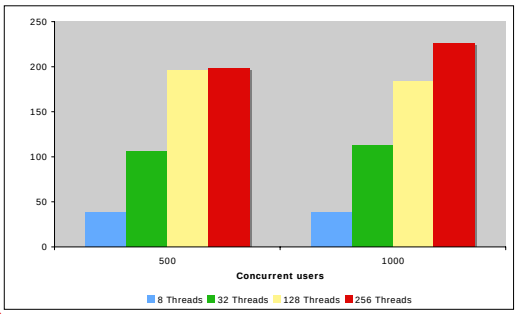
Tests Results



Tomcat threads

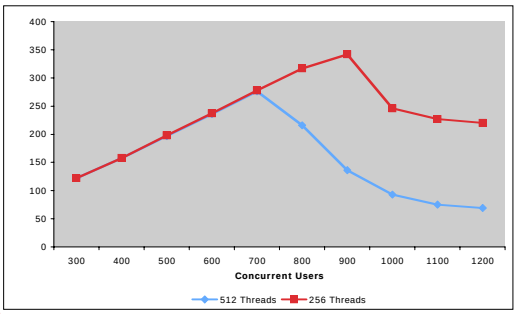
- JBoss AS processes each web request in a separate Tomcat thread
- Tomcat maintains a pool of threads to avoid thread creation / destruction overhead
  - ✓ Too few: Cannot utilize the full CPU due to insufficient parallelization (CPU runs < 30% at peak throughput)
  - ✓ Too many: The thread context switching adds too much CPU overhead
  - ✓ The more CPU you have, the more threads you need (we have 8 logical CPUs)
- Configured in deploy/jbossweb-tomcat55.sar/server.xml file
  - ✓ In the HTTP Connector
  - ✓ maxThreads specifies the size of thread pool
  - ✓ accept specifies how many requests are allowed in queue

Too few threads



37

Too many threads



256 threads is the optimal number we choose

38

Agenda

- The basics
- The test tools and environment
- Interpret performance tests results
- Optimize Seam on a single server
- Load balanced cluster
- In the real world

39

No state replication

- Use sticky session in mod_jk
  - ✓ Requests from the same session are always forwarded to the same server node
- The servers are installed as stand-alone servers (EJB3 w/o clustering profile)
- No changes to the application
- No fail-over
- Scales linearly with number of nodes until
  - ✓ the network saturates
  - ✓ the load balancer is overloaded
- Great for balancing multiple virtual servers on a blade

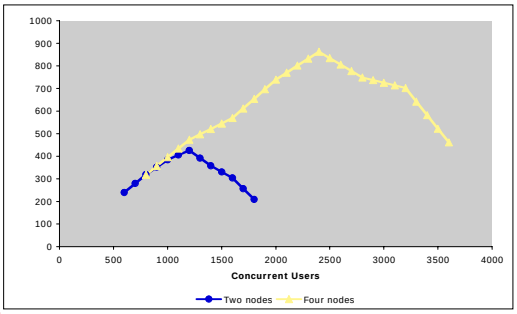
40

mod_jk tips

- The Apache maxThreads must match the sum of Tomcat maxThreads (allow 20% error margin)
- Turn off “keep-alive” connection in Apache during testing
- Use non-prefork Apache MPM if possible

41

Tests Results



42

Failover support

- State information is replicated to failover nodes
  - ✓ In Seam, HTTP session is only replicated at creation and timeout
  - ✓ The stateful session bean is replicated throughout the conversation
- Replication is slow
  - ✓ Serialization is CPU intensive
  - ✓ Network is slower than RAM
  - ✓ Overhead increases geometrically with number of failover nodes

SFSB replication support is still under active development. No test data is officially published here.

43

Must use sticky session

- Simple load balancing without sticky session distributes requests randomly to all nodes in the cluster
  - ✓ Works under low load (browser test)
  - ✓ At 30 TPS, we see around 10% of the requests generate HTTP 500 errors
- Sticky session
  - ✓ Each node handles its own sessions and replicates the states over other nodes in the cluster as failover
  - ✓ Config mod_jk to retry failover nodes

44

The scalability problem

Replicating the state of each node to all nodes in the cluster is not scalable: the replication work load increases geometrically with the number of nodes in the cluster.

45

Buddy replication

- New in JBoss Cache 1.4
  - ✓ Available in JBoss 5.0 alpha
  - ✓ Will be back ported to JBoss 4.0.5
- Each node only replicates to a buddy failover node
- mod_jk is clever enough to know which failover node to retry if a session dies
- Tested on un-optimized alpha code
  - ✓ Two buddy nodes achieve performance of 1 node w/o failover
  - ✓ Expect huge performance improvement over the next several weeks

46

Further thinking

- Asymmetric failover
  - ✓ One node in the buddy pair acts as the failover and it does not serve requests unless the other fails
  - ✓ Better user experience when failover
  - ✓ Redundant hardware needed
- N+1 failover
  - ✓ One node acts as the failover node for N nodes
  - ✓ The failover node is a “buddy” to all other nodes in the cluster
  - ✓ May need N+2, N+3 … depending on load
  - ✓ Good user experience when failover occurs

47

Agenda

- The basics
- The test tools and environment
- Interpret performance tests results
- Optimize Seam on a single server
- Load balanced cluster
- In the real world

48

Database access is the real bottleneck

- A single Dell PowerEdge 1855 Blade has enough power to run a Seam application for a large community
  - ✓ 20 million web transactions / day (250 TPS)
  - ✓ > 5,000 concurrent users
  - ✓ A simple cluster can support more than 10,000 concurrent users
- Database access overload if each web tx requires a database roundtrip
  - ✓ O/R mapping, JDBC connections will be overwhelmed
  - ✓ Seam extended persistence context makes it easy to reduce database round trips


The bottom line

Seam business logic components run sufficiently fast on today's state of the art hardware.

Your application's overall throughput is more likely to be limited by the data access layer.


Next steps

- Test performance of multiple VMs on a single blade
- Test buddy replicated clusters
- Add database access layer to the application
- Test replicated database second level cache


Q&A