
Performance factors in JBoss 3.2.0

The effects of simple coding optimizations

Author: Jon Barnett

Document version: 1.2

Table of contents

1	Effects of optimizing JBoss 3.2.0 code	1-1
1.1	Optimizing the JBoss container.....	1-1
1.2	Test setup	1-1
1.2.1	JBoss 3.2.0 configuration	1-1
1.2.2	Jetty 4.2.9 configuration	1-1
1.2.3	Postgresql connection pool	1-1
1.2.4	Test applications	1-2
1.2.5	Test clients.....	1-2
1.2.6	Test environment	1-2
1.3	Measurements and calculations.....	1-3
1.4	Container measurement results	1-4
1.4.1	Default JBoss 3.2.0.....	1-4
1.4.2	Optimized JBoss 3.2.0.....	1-4
1.4.3	Change set 1: optimized JBoss 3.2.0	1-4
1.4.4	Change set 2: optimized JBoss 3.2.0	1-5
1.4.5	Change set 3: optimized JBoss 3.2.0	1-5
1.4.6	Change set 4: optimized JBoss 3.2.0	1-5
1.4.7	Change set 5: optimized JBoss 3.2.0	1-6
1.4.8	Change set 6: optimized JBoss 3.2.0	1-6
1.4.9	Change set 7: optimized JBoss 3.2.0	1-7
1.4.10	Change set 8: optimized JBoss 3.2.0	1-7
1.4.11	Change set 8a: optimized JBoss 3.2.0.....	1-7
1.5	Application measurement results	1-8
1.5.1	Default JBoss 3.2.0.....	1-8
1.5.2	Optimized JBoss 3.2.0.....	1-8
1.5.3	Change set 1: optimized JBoss 3.2.0	1-8
1.5.4	Change set 2: optimized JBoss 3.2.0	1-8
1.5.5	Change set 3: optimized JBoss 3.2.0	1-8
1.5.6	Change set 4: optimized JBoss 3.2.0	1-8
1.5.7	Change set 5: optimized JBoss 3.2.0	1-8
1.5.8	Change set 6: optimized JBoss 3.2.0	1-8
1.5.9	Change set 7: optimized JBoss 3.2.0	1-9
1.5.10	Change set 8: optimized JBoss 3.2.0	1-9
1.5.11	Change set 8a: optimized JBoss 3.2.0.....	1-9
1.6	Comparison of container performance	1-9
1.7	Comparison of J2EE application performance.....	1-10
1.8	Discussion	1-11
A	Collection measurements using Trove benchmarks	A-1
A.1	Timing benchmark code	A-1
A.2	Linux IBM SDK timing benchmark results.....	A-5
A.3	Linux Sun JDK timing benchmark results.....	A-6
A.4	Linux IBM SDK memory benchmark results.....	A-8
A.5	Linux Sun SDK memory benchmark results.....	A-9
B	Container test code.....	B-1
B.1	Stateless session bean implementation	B-1
B.2	Test load client code.....	B-3

1 Effects of optimizing JBoss 3.2.0 code

This study is intended to examine the effects of coding practices to improve the performance of JBoss 3.2.0. It is still a worthwhile exercise to examine the performance of your own applications and identify any possible bottlenecks within your own architecture that would nullify any performance gains before looking at the JBoss microkernel coding.

1.1 Optimizing the JBoss container

The default JBoss 3.2.0 binaries as distributed from the JBoss website has no bytecode optimizations. Optimizing your bytecode, at least in theory, should improve the speed at which your code executes by removing extraneous instructions and improving the organization of bytecode sequences. The optimizations should also reduce the memory footprint of classes.

From our previous studies, we have found that these optimizations and the use of performance JVMs can improve performance. Beyond that study, we now look to the use of performance coding to boost results. We examined this in light of studies in Java code performance in the areas of the Java Collection classes and in particular, the Trove collection implementations. Appendix A contains measurements from the benchmark relevant to the JBoss code optimizations. The JBoss architecture makes heavy use of the Collection classes to cache information necessary for the J2EE infrastructure. However, the Trove benchmarks show that the standard Collection implementations deliver less than optimal results in most cases. This study has the aim of examining small changes to the code to implement better Collection use and that avoids major rewriting of classes. We will attempt to measure the effect of these changes to the infrastructure performance, and at a high level, the effect on applications.

1.2 Test setup

1.2.1 JBoss 3.2.0 configuration

The JBoss 3.2.0 configuration used in the tests is the default runtime instance. This is largely unchanged except for the removal of the JMX console web application, the UUID key generator and the scheduler. All environments contain the Amity framework components and the Amity suite of products, although none of these products were used during the testing.

1.2.2 Jetty 4.2.9 configuration

The standard embedded Jetty service for JBoss 3.2.0 was used. However, we compiled the IBM JSSE listener and the socket channel listener to make use of the more scalable standard listener and a secure listener for the IBM JVM. The secure socket listeners were configured but not used in the tests.

1.2.3 Postgresql connection pool

The connection pool was configured so that no upper limit was created for the pool that would throttle the application load in any way. However, an optimized Postgresql JDBC driver was used in the optimized JBoss 3.2.0 container whereas the default Postgresql JDBC driver was used with the default JBoss 3.2.0 container.

1.2.4 Test applications

The majority of the testing is performed using a stateless session bean. The aim of these set of tests is to detect any change in the performance of the EJB container as we are changing the interface code rather than any fundamental change in the execution of code within EJBs.

For this purpose, the execution code for the primary method involves a simple counter increment. We could also have implemented this method with no operation but for the purposes of ensuring that parameters are passed correctly between client and EJB we have included the operation. The operation is sufficiently small as to contribute negligibly to the performance cost.

We also use the stateless session bean to invoke the method from within the VM using both the local and remote EJB interface to measure the in-VM performance. The stateless session bean code has been included in Appendix B.

The tests are only performed against a stateless session bean, but provide what we hope is an indicative measure as most optimizations are centred on the common interactions with the JBoss infrastructure. We test the lookup performance, and the creation and invocation performance. We do this in an in-VM situation and also a remote situation, although for the latter we avoid transmissions over actual physical network media. Multiple threads generate what we hope are a better load than a pure serial set of calls.

We retain the application test from the previous study on bytecode optimizations to observe the effect of the current changes on an application. The application, as before, makes use of the JBoss infrastructure consists of a single page of content. The content is created and assembled through four servlets. These servlets access four data tables in Postgresql to obtain the content.

The application byte code has been optimized, as has the application architecture. These optimizations have remained in place for all tests, to ensure that the application itself would operate as fast as possible.

1.2.5 Test clients

For the container measurements, we wrote a small multi-threaded client that connects to the stateless session bean. Originally we had performed measurements with a heterogenous mixture of calls to the JBoss infrastructure. However, since the calls were competing for process access, the readings for a particular set of measurements were more unstable for short iteration averages. We switched the testing so that measurements are taken for a set of competing homogenous calls.

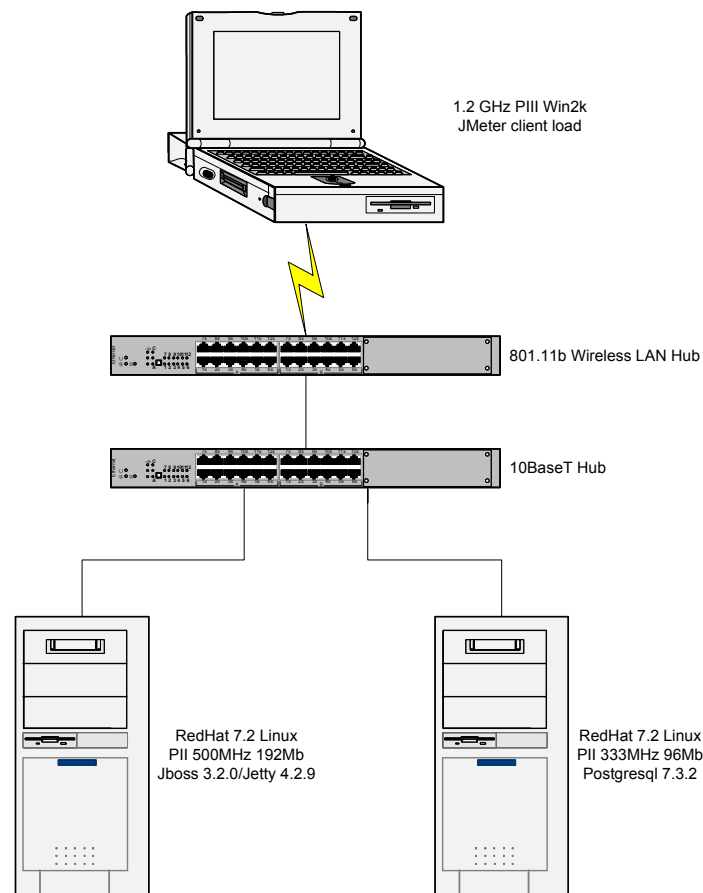
The individual calls are performed in 10,000 block sets so that a reasonable time can be measured for the block to calculate the averages for each call. Appendix B contains the client code.

We use Apache JMeter to generate the application test load. While we want the load to have sufficient weight, we do not want to create so many requests that the system under test cannot service the load. For our purposes, we decided that 3 requests per second with 48 clients would provide a reasonable test load. This was generated by defining 48 threads, using a uniform random timer with a maximum delay of 30 seconds. Only the servlet content was retrieved, excluding any graphic images, style sheets or other linked content. A large sample space was taken to ensure that statistics generated were not unduly biased by network fluctuations or other events.

We accessed the application using the socket channel listener so that any differences between the IBM JSSE listener and the Sun JSSE listener had no bearing on the results.

1.2.6 Test environment

The test environment features a fairly modest server environment with some fixed delays. Improvements in the network speeds and the database server would reduce these fixed delays to the response time and most likely would amplify the relative magnitude of the performance increases measured. However, the measurements are intended to give an indication of the possible performance improvements rather than being an absolute measure.



1.3 Measurements and calculations

The container interaction measurements are taken from a pair of parallel clients, each generating a block of 10,000 serial calls. The maximum measurement is taken as the approximation of the time for both threads to complete which is the time for 20,000 calls to be completed. The call time averages are computed on this information.

The results from JMeter are reported directly and we compare this with results taken from our previous study.

All data measurements are in milliseconds and were taken in a steady state configuration so creation times for components are excluded from the values. The IBM SDK 1.4.0 was used as the JVM as this was previously determined as the best performing virtual machine. Memory readings were taken after the container tests were performed but before the application tests were performed. These give an indicative measure of the memory that appears to be used from the operating system perspective. Over time the physical memory usage will reduce when the garbage collection routines in the server will reclaim unused memory. However, for the current round of testing, we did not wish to impact the response measurements by imposing either memory monitoring or enforced memory management. These may be included at some later date.

1.4 Container measurement results

1.4.1 Default JBoss 3.2.0

The JBoss code is not optimized. No changes have been made to the code.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
401	471	3038	3239	4773	4960	190059	191205	291018	292034
362	515	2927	3412	3656	3977	186367	188503	288126	289676
945	1221	2843	3007	3851	4096	181150	182886	288868	291860
389	969	2259	2388	3223	3259	195200	196563	285201	285921
1095	1219	2140	3021	3704	3734	185507	186048	287112	287389
	879		3013		4005		189041		289376

VSZ	RSS
199988 kb	136852 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
44 μ s	151 μ s	200 μ s	9.45 ms	14.47 ms

1.4.2 Optimized JBoss 3.2.0

The JBoss code is optimized. No changes have been made to the code.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
407	576	2303	2460	3286	3339	185491	185624	274030	276433
912	949	2159	3055	3544	3797	184946	185308	277421	277916
362	467	2196	2338	3442	3874	181569	183207	278014	279459
822	1040	2256	2423	4127	4276	180812	181852	278777	280391
351	466	3065	3088	3249	3306	184970	185355	279544	280910
	700		2673		3718		184269		279022

VSZ	RSS
177104 kb	127360 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
35 μ s	134 μ s	186 μ s	9.21 ms	13.95 ms

1.4.3 Change set 1: optimized JBoss 3.2.0

The JBoss code is optimized. The method `getChildrenByTagName` has been replaced with a similar function that returns `ArrayList` rather than a `Collection`. Dependent methods are modified to suit.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
396	415	2729	2748	3452	3554	180213	181862	277321	278347
908	937	2213	2467	3092	3128	187870	188182	277919	279469
324	409	2034	2309	3043	3130	180836	180846	278014	279459
854	861	2654	2690	2917	3699	181640	181962	276201	280847
344	401	2006	2277	3085	3447	186684	188305	275870	276390
	605		2498		3392		184231		278902

VSZ	RSS
177600 kb	126108 kb

In VM			External		
Lookup	Local	Remote	Lookup	Remote	
30 μ s	125 μ s	170 μ s	9.21 ms	13.95 ms	

1.4.4 Change set 2: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, the marshalled invocation mapping uses TLongObjectHashMap and the container interface hashes use TObjectLongHashMap.

In VM						External				
Lookup		Local		Remote		Lookup		Remote		
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
351	427	2090	2273	2848	2909	181516	182158	278295	279175	
349	422	2111	2136	2836	3013	183493	185421	279531	280611	
798	1031	3197	3887	2728	3019	180994	181410	279908	280058	
336	439	2125	2350	2720	3041	185724	186280	283371	283833	
874	972	2116	2264	3398	3670	182081	182470	275870	276390	
	658		2582		3130		183548		280013	

VSZ	RSS
175284 kb	126304 kb

In VM			External		
Lookup	Local	Remote	Lookup	Remote	
33 μ s	129 μ s	156 μ s	9.18 ms	14.00 ms	

1.4.5 Change set 3: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, TLongObjectHashMap is used in the proxy factory.

In VM						External				
Lookup		Local		Remote		Lookup		Remote		
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
355	481	2632	2842	2799	2830	182448	182475	278295	279175	
876	892	1964	2236	2723	2943	186811	187721	279531	280611	
298	420	2050	2346	3208	3519	183861	185204	279908	280058	
862	874	2588	2753	2749	3105	184751	185435	283371	283833	
363	393	2031	2255	2761	3077	183416	184787	281886	283251	
	612		2486		3095		185124		281386	

VSZ	RSS
177732 kb	123396 kb

In VM			External		
Lookup	Local	Remote	Lookup	Remote	
31 μ s	124 μ s	155 μ s	9.26 ms	14.07 ms	

1.4.6 Change set 4: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, TLongObjectHashMap is used in the common methodHash and NamingContext/NamingServer uses an ArrayList instead of a Vector/List.

In VM						External				
Lookup		Local		Remote		Lookup		Remote		
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	
869	874	2058	2079	2883	2942	182448	182701	280470	281946	
385	423	1951	2179	2876	2959	186811	183211	284415	285183	
287	355	2738	2786	2674	3005	183861	184590	278374	279040	
321	435	1866	2142	2897	2882	184751	191477	279798	281185	
853	865	1905	2117	3262	3611	183416	182133	274954	277063	
	590		2261		3080		184822		280883	

VSZ	RSS
177120 kb	122560 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
30 μ s	113 μ s	154 μ s	9.24 ms	14.04 ms

1.4.7 Change set 5: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, TLongObjectHashMap is used in the common methodHash and NamingContext/NamingServer uses an ArrayList instead of a Vector/List.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
363	433	2127	2132	2591	3024	185567	186251	279696	282713
395	470	2113	2144	2683	3022	184052	184134	280596	280942
230	436	2574	2682	2566	3614	181873	182946	282319	283049
888	895	1996	2272	2674	2914	187193	187671	277414	281159
423	510	2077	2084	2784	2825	183345	183747	276939	279936
	549		2263		3080		184950		281560

VSZ	RSS
178868 kb	121488 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
27 μ s	113 μ s	154 μ s	9.25 ms	14.08 ms

1.4.8 Change set 6: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, TLongObjectHashMap is used in the common methodHash and NamingContext/NamingServer uses an ArrayList instead of a Vector/List.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
295	429	2185	2190	3059	3119	184314	185334	281135	281136
307	432	2066	2290	3599	3610	181866	183319	283812	283892
809	836	2150	2790	3015	3015	183328	183968	281273	281737
269	430	2086	2140	3094	3111	182255	184326	280217	281179
295	437	2132	2191	3042	3094	183511	186944	280311	280944
	513		2320		3190		184778		281778

VSZ	RSS
178940 kb	122616 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
26 μ s	116 μ s	160 μ s	9.24 ms	14.09 ms

1.4.9 Change set 7: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, TLinkedList is used for txWaitQueue in QueuedPessimisticEJBLock and GlobalTXEntityMap vector entities are replaced with an ArrayList for faster iteration and access.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
341	429	1773	2140	2917	2923	177080	179886	276775	277830
278	422	1836	2093	3203	3241	179148	180531	277994	279206
267	397	2371	2548	2868	2891	183270	183292	277445	278372
371	428	1843	2090	2835	2853	189133	189225	277698	277911
832	961	1840	2107	2821	2893	182795	183181	286422	286561
	527		2196		2960		183223		279976

VSZ	RSS
176404 kb	122204 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
26 μ s	110 μ s	148 μ s	9.16 ms	14.00 ms

1.4.10 Change set 8: optimized JBoss 3.2.0

The JBoss code is optimized. A cumulative code change, getMethodParams, getCMPFields, convertToJavaClasses and get* from BeanMetaData.java return the underlying ArrayLists that they use for faster iteration and access than Collection/Iterator.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
261	401	1954	1963	2696	2841	179578	180706	282907	283151
302	432	1725	1992	2829	2915	178561	178814	279745	280546
334	423	2253	2345	2714	2758	178717	179940	280334	281637
353	454	1728	2062	3151	3342	182353	183356	275882	276938
950	952	2361	2419	2780	2800	179191	179466	277995	278162
	532		2156		2931		180456		280087

VSZ	RSS
176404 kb	122204 kb

In VM			External	
Lookup	Local	Remote	Lookup	Remote
27 μ s	108 μ s	147 μ s	9.03 ms	14.00 ms

1.4.11 Change set 8a: optimized JBoss 3.2.0

The JBoss code is optimized. The IBM SDK was upgraded to 1.4.1.

In VM						External			
Lookup		Local		Remote		Lookup		Remote	
Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
257	457	1835	1935	2661	2673	180215	181167	271872	273870
256	450	1951	1961	2638	2772	170969	170994	270923	271326
319	381	1953	2503	2649	2888	176400	177084	272559	275460
335	987	1805	2049	2668	3308	172032	173453	259137	262295
370	437	1889	2125	2578	2869	176713	177369	272378	273060
	542		2115		2902		176013		271202

VSZ	RSS
139992 kb	114888 kb

Lookup	In VM		External	
	Local	Remote	Lookup	Remote
27 μ s	106 μ s	145 μ s	8.80 ms	13.56 ms

1.5 Application measurement results

1.5.1 Default JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	155 ms	60 ms	16694 ms	2.7 / sec	9	10

1.5.2 Optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	125 ms	50 ms	2734 ms	2.8 / sec	7	8

1.5.3 Change set 1: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	122 ms	50 ms	3011 ms	2.8 / sec	7	8

1.5.4 Change set 2: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	118 ms	60 ms	2836 ms	2.8 / sec	7	8

1.5.5 Change set 3: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	116 ms	60 ms	2784 ms	2.8 / sec	7	8

1.5.6 Change set 4: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	111 ms	60 ms	2594 ms	2.9 / sec	7	8

1.5.7 Change set 5: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	112 ms	60 ms	2904 ms	2.9 / sec	7	8

1.5.8 Change set 6: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	110 ms	50 ms	2854 ms	2.9 / sec	7	8

1.5.9 Change set 7: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	109 ms	51 ms	2444 ms	2.9 / sec	7	8

1.5.10 Change set 8: optimized JBoss 3.2.0

Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	108 ms	50 ms	2463 ms	2.9 / sec	7	8

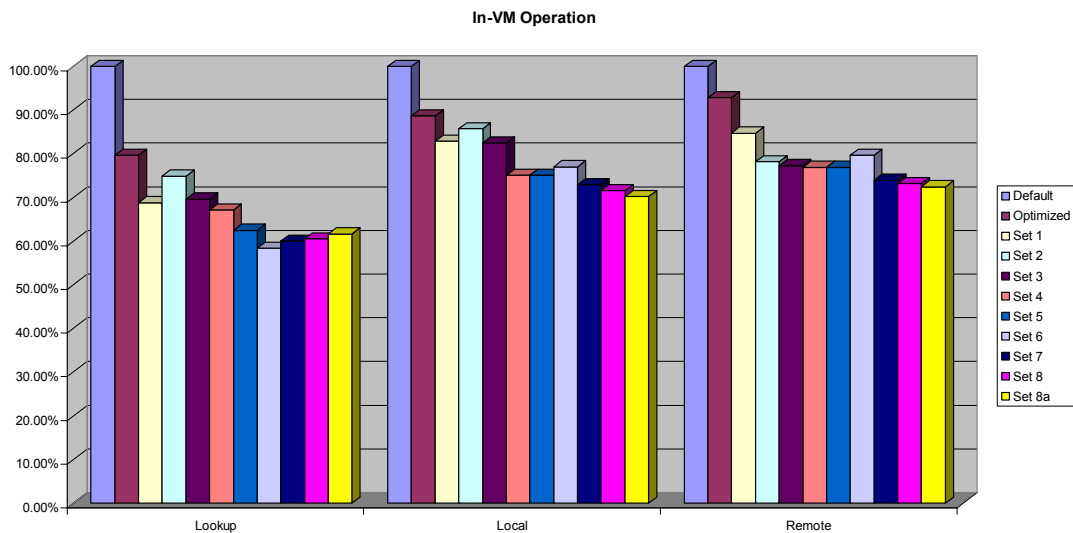
1.5.11 Change set 8a: optimized JBoss 3.2.0

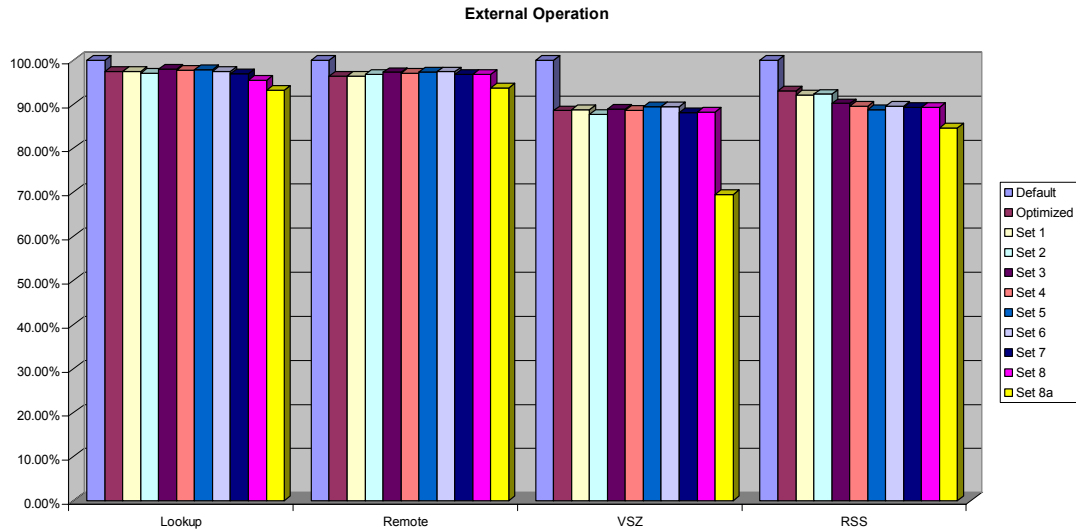
Samples	Request response			Request rate	Connection pool	
	Average	Minimum	Maximum		Average	Maximum
9600	105 ms	50 ms	2405 ms	2.9 / sec	7	8

1.6 Comparison of container performance

Using the default JBoss code and non-optimized bytecode compilation as the reference, the following comparative values are tabled.

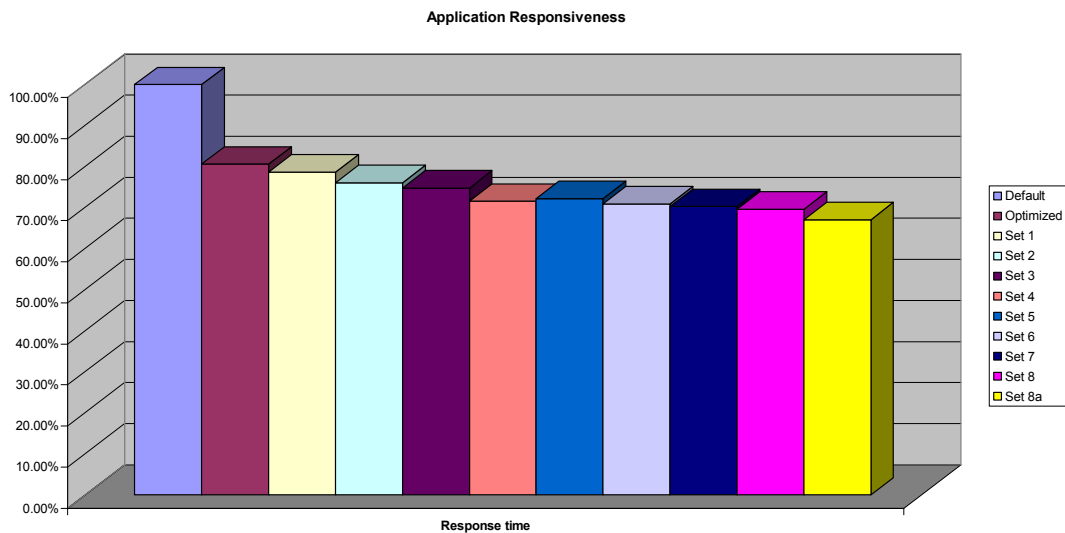
JBoss 3.2.0	In VM			External	
	Lookup	Local	Remote	Lookup	Remote
Default	44 μ s	151 μ s	200 μ s	9.45 ms	14.47 ms
Optimized	79.55%	88.74%	93.30%	97.46%	96.41%
Change set 1	68.83%	82.91%	78.15%	97.46%	96.38%
Change set 2	74.86%	85.70%	78.15%	97.09%	96.76%
Change set 3	69.62%	82.51%	77.28%	97.93%	97.24%
Change set 4	67.12%	75.04%	76.90%	97.77%	97.07%
Change set 5	62.46%	75.11%	76.90%	97.84%	97.30%
Change set 6	58.36%	77.00%	79.65%	97.74%	97.37%
Change set 7	59.95%	72.88%	73.91%	96.92%	96.95%
Change set 8	60.52%	71.56%	73.18%	95.46%	96.79%
Change set 8a	61.66%	70.20%	72.46%	93.11%	93.72%





1.7 Comparison of J2EE application performance

JBoss 3.2.0	Application response
Default	155 ms
Optimized	80.65%
Change set 1	78.71%
Change set 2	76.13%
Change set 3	74.84%
Change set 4	71.61%
Change set 5	72.26%
Change set 6	70.97%
Change set 7	70.32%
Change set 8	69.68%
Change set 8a	67.10%



1.8 Discussion

From the results gathered to date, the simplest and best gain for application performance is to perform bytecode optimizations on the JBoss code. This can reduce the response time by up to 20 percent for our test application. Through some simple coding changes, mainly involving the optimal use of collection classes, we can obtain a further 11 percent reduction. The simplicity of the implementation makes the gains a worthwhile effort.

We also note that the current changes have little impact on remote clients as the lookup and invocation times are significantly larger than the improvements achieved. However, the changes do reduce lookup and invocation times in the local VM by up to 40 percent and 30 percent respectively. This translates into reduced lag between content requests and content delivery making the web site more responsive. These reductions will also increase the load capacity of the system because of the reduction in resource consumption.

One other point of interest is the significant difference between the internal VM calls and the external VM calls. The internal method invocation time, accepting that the EJB execution time is negligible in the test instance, is 100 times faster than the external invocation. The internal JNDI lookup is 200 times faster than the external lookup. This would indicate that for best response and optimal use of resources, the embedded Tomcat and Jetty containers should be utilised rather than reverting to standalone Servlet container configurations.

Finally, with the results gathered, there appears to be periods in which invocations and lookups are blocked. This is indicated by the measurements for the in-VM calls where there are some measurements that are noticeably different to the others. The application testing also indicates the same as the response measurements form around a well defined median with the occasional reading significantly deviating from this. As yet, we have not determined the source of this irregularity.

A Collection measurements using Trove benchmarks

We use the Trove benchmark program to measure particular performance characteristics of Java Collections. In particular, we have added some tests that also focus on performance between various types of native Collections.

We also use Trove for some specific JBoss code changes because of the performance. We haven't provided any specific code for memory tests but supply instead the results from the standard Trove memory tests.

A.1 Timing benchmark code

We provide only the custom code we have implemented in the `gnu.trove.benchmark.Main`.

```
////////////////////////////////////
// Custom routines

static Operation getArrayListGetOp() {
    final ArrayList theirMap = new ArrayList(dataset.size());
    final TIntObjectHashMap ourMap = new TIntObjectHashMap(dataset.size());
    int j = 0;
    for (Iterator i = dataset.iterator(); i.hasNext(); j++) {
        Object o = i.next();
        theirMap.add(o);
        ourMap.put(j, o);
    }

    return new Operation() {
        public final void theirs() {
            ArrayList m = theirMap;
            Object o;
            for (int size = m.size(); size-- > 0;) {
                o = m.get(size);
            }
        }

        public final void ours() {
            TIntObjectHashMap m = ourMap;
            // Iterator i = m.keySet().iterator();
            Object o;
            for (int size = m.size(); size-- > 0;) {
                o = m.get(size);
            }
        }

        public String toString() {
            return "compares ArrayList and TIntObjectHash get(i) over "
                + dataset.size() + " map keys";
        }

        public int getIterationCount() {
            return 10;
        }
    };
}

static Operation getLongObjectGetOp() {
    final HashMap theirMap = new HashMap(dataset.size());
    final TLongObjectHashMap ourMap = new TLongObjectHashMap(dataset.size());
    long j = 0;
    for (Iterator i = dataset.iterator(); i.hasNext(); j++) {
        Object o = i.next();
        theirMap.put(new Long(j), o);
        ourMap.put(j, o);
    }
}
```

```

return new Operation() {
    public final void theirs() {
        Map m = Collections.unmodifiableMap(theirMap);
        Object o;
        for (int size = m.size(); size-- > 0;) {
            o = m.get(new Long(size));
        }
    }

    public final void ours() {
        TLongObjectHashMap m = ourMap;
        // Iterator i = m.keySet().iterator();
        Object o;
        for (int size = m.size(); size-- > 0;) {
            o = m.get(size);
        }
    }

    public String toString() {
        return "compares HashMap and TLongObjectHashMap get() over "
            + dataset.size() + " map keys";
    }

    public int getIterationCount() {
        return 10;
    }
};

}

static Operation getHashMapArrayGetOp() {
    final HashMap theirMap = new HashMap(dataset.size());
    final ArrayList ourMap = new ArrayList(dataset.size());
    long j = 0;
    for (Iterator i = dataset.iterator(); i.hasNext(); j++) {
        Object o = i.next();
        theirMap.put(new Long(j), o);
        ourMap.add(o);
    }

    return new Operation() {
        public final void theirs() {
            HashMap m = theirMap;
            Iterator i = m.keySet().iterator();
            Object o;
            while (i.hasNext()) {
                o = i.next();
            }
        }

        public final void ours() {
            ArrayList m = ourMap;
            // Iterator i = m.keySet().iterator();
            Object o;
            int j = m.size();
            for (int i = 0; i < j; i++) {
                o = m.get(i);
            }
        }

        public String toString() {
            return "compares HashMap and ArrayList retrieval over "
                + dataset.size() + " map keys";
        }

        public int getIterationCount() {
            return 10;
        }
    };
}

```

```

static Operation getIncrementArrayGetOp() {
    final ArrayList theirMap = new ArrayList(dataset.size());
    final ArrayList ourMap = new ArrayList(dataset.size());
    for (Iterator i = dataset.iterator(); i.hasNext();) {
        Object o = i.next();
        theirMap.add(o);
        ourMap.add(o);
    }

    return new Operation() {
        public final void theirs() {
            ArrayList m = theirMap;
            Object o;
            for (int i = 0; i < m.size(); i++) {
                o = m.get(i);
            }
        }

        public final void ours() {
            ArrayList m = ourMap;
            // Iterator i = m.keySet().iterator();
            Object o;
            int j = m.size();
            for (int i = 0; i < j; i++) {
                o = m.get(i);
            }
        }

        public String toString() {
            return "compares ArrayList retrieval using size() over "
                + dataset.size() + " map keys";
        }

        public int getIterationCount() {
            return 10;
        }
    };
}

static Operation getNewObjectArrayGetOp() {
    final ArrayList theirMap = new ArrayList(dataset.size());
    final ArrayList ourMap = new ArrayList(dataset.size());
    for (Iterator i = dataset.iterator(); i.hasNext();) {
        Object o = i.next();
        theirMap.add(o);
        ourMap.add(o);
    }

    return new Operation() {
        public final void theirs() {
            ArrayList m = theirMap;
            for (int i = 0; i < m.size(); i++) {
                Object o = m.get(i);
            }
        }

        public final void ours() {
            ArrayList m = ourMap;
            // Iterator i = m.keySet().iterator();
            Object o;
            int j = m.size();
            for (int i = 0; i < j; i++) {
                o = m.get(i);
            }
        }

        public String toString() {
            return "compares ArrayList retrieval using and size() over "
                + dataset.size() + " map keys";
        }

        public int getIterationCount() {
            return 10;
        }
    };
}

```

```

static Operation getObjectLongGetOp() {
    final HashMap theirMap = new HashMap(dataset.size());
    final TObjectLongHashMap ourMap = new TObjectLongHashMap(dataset.size());
    long j = 0;
    for (Iterator i = dataset.iterator(); i.hasNext(); j++) {
        Object o = i.next();
        theirMap.put(new Long(j), new Long(j));
        ourMap.put(new Long(j), j);
    }

    return new Operation() {
        public final void theirs() {
            HashMap m = theirMap;
            long k;
            for (long j = m.size(); j-- > 0; ) {
                k = ((Long)m.get(new Long(j))).longValue();
            }
        }

        public final void ours() {
            TObjectLongHashMap m = ourMap;
            // Iterator i = m.keySet().iterator();
            Object o;
            long k;
            for (long j = m.size(); j-- > 0; ) {
                k = m.get(new Long(j));
            }
        }

        public String toString() {
            return "compares TObjectLongHashMap over " + dataset.size()
                + " map keys";
        }

        public int getIterationCount() {
            return 10;
        }
    };
}

static Operation getArrayIteratorGetOp() {
    final ArrayList theirMap = new ArrayList(dataset.size());
    final ArrayList ourMap = new ArrayList(dataset.size());
    for (Iterator i = dataset.iterator(); i.hasNext(); ) {
        Object o = i.next();
        theirMap.add(o);
        ourMap.add(o);
    }

    return new Operation() {
        public final void theirs() {
            Collection m = theirMap;
            Iterator i = m.iterator();
            Object o;
            while (i.hasNext()) {
                o = i.next();
            }
        }

        public final void ours() {
            ArrayList m = ourMap;
            Object o;
            int j = m.size();
            for (int i = 0; i < j; i++) {
                o = m.get(i);
            }
        }

        public String toString() {
            return "compares ArrayList get() against Collection" +
                " hasNext()/next() over " + dataset.size() + " map keys";
        }
    };
}

```

```

        public int getIterationCount() {
            return 10;
        }
    };
}

```

A.2 Linux IBM SDK timing benchmark results

```
-----
GNU Trove Benchmark suite
-----
```

```
java.vm.name=Classic VM
java.runtime.version=1.4.0
os.name=Linux
os.arch=x86
os.version=2.4.18
-----
```

```
compares 100000 Set.contains() operations. 3 are actually present in set
Iterations: 10
Their total (msec): 950
Our total (msec): 1964
Their average (msec): 95
Our average (msec): 196
-----
```

```
sums a 100000 element Set of Integer objects. Their approach uses
Iterator.hasNext()/next(); ours uses THashSet.forEach(TObjectProcedure)
Iterations: 10
Their total (msec): 705
Our total (msec): 236
Their average (msec): 70
Our average (msec): 23
-----
```

```
compares Iterator.hasNext()/ Iterator.next() over 100000 keys
Iterations: 10
Their total (msec): 501
Our total (msec): 472
Their average (msec): 50
Our average (msec): 47
-----
```

```
compares Iterator.next() over 100000 map keys
Iterations: 10
Their total (msec): 460
Our total (msec): 270
Their average (msec): 46
Our average (msec): 27
-----
```

```
compares 100000 LinkedList.add() operations
Iterations: 10
Their total (msec): 762
Our total (msec): 580
Their average (msec): 76
Our average (msec): 58
-----
```

```
100000 entry primitive int map.put timing run; no basis for comparison
Iterations: 10
Their total (msec): 0
Our total (msec): 675
Their average (msec): 0
Our average (msec): 67
-----
```

```
compares 100000 Map.put() operations
Iterations: 10
Their total (msec): 2365
Our total (msec): 906
Their average (msec): 236
Our average (msec): 90
-----
```

```
compares 20000 Set.contains() operations
Iterations: 10
Their total (msec): 240
Our total (msec): 215
Their average (msec): 24
Our average (msec): 21
-----
```

```
compares 100000 Map.get() operations
Iterations: 10
Their total (msec): 731
Our total (msec): 767
Their average (msec): 73
Our average (msec): 76
-----
compares 100000 Set.add() operations
Iterations: 10
Their total (msec): 2351
Our total (msec): 769
Their average (msec): 235
Our average (msec): 76
-----
compares ArrayList and TIntObjectHash get(i) over 100000 map keys
Iterations: 10
Their total (msec): 45
Our total (msec): 224
Their average (msec): 4
Our average (msec): 22
-----
compares HashMap and TLongObjectHashMap get() over 100000 map keys
Iterations: 10
Their total (msec): 1583
Our total (msec): 287
Their average (msec): 158
Our average (msec): 28
-----
compares HashMap and ArrayList retrieval over 100000 map keys
Iterations: 10
Their total (msec): 547
Our total (msec): 47
Their average (msec): 54
Our average (msec): 4
-----
compares ArrayList retrieval using size() over 100000 map keys
Iterations: 10
Their total (msec): 60
Our total (msec): 46
Their average (msec): 6
Our average (msec): 4
-----
compares ArrayList retrieval using new object and size() over 100000 map keys
Iterations: 10
Their total (msec): 59
Our total (msec): 47
Their average (msec): 5
Our average (msec): 4
-----
compares ArrayList get() against Collection hasNext()/next() over 100000 map keys
Iterations: 10
Their total (msec): 116
Our total (msec): 47
Their average (msec): 11
Our average (msec): 4
-----
```

A.3 Linux Sun JDK timing benchmark results

```
-----
GNU Trove Benchmark suite
-----
java.vm.name=Java HotSpot(TM) Client VM
java.runtime.version=1.4.1_01-b01
os.name=Linux
os.arch=i386
os.version=2.4.18
-----
compares 100000 Set.contains() operations. 3 are actually present in set
Iterations: 10
Their total (msec): 950
Our total (msec): 1787
Their average (msec): 95
Our average (msec): 178
-----
```

```
sums a 100000 element Set of Integer objects. Their approach uses
Iterator.hasNext()/next(); ours uses THashSet.forEach(TObjectProcedure)
Iterations: 10
Their total (msec): 779
Our total (msec): 261
Their average (msec): 77
Our average (msec): 26
-----
compares Iterator.hasNext()/ Iterator.next() over 100000 keys
Iterations: 10
Their total (msec): 529
Our total (msec): 449
Their average (msec): 52
Our average (msec): 44
-----
compares Iterator.next() over 100000 map keys
Iterations: 10
Their total (msec): 523
Our total (msec): 216
Their average (msec): 52
Our average (msec): 21
-----
compares 100000 LinkedList.add() operations
Iterations: 10
Their total (msec): 2893
Our total (msec): 605
Their average (msec): 289
Our average (msec): 60
-----
100000 entry primitive int map.put timing run; no basis for comparison
Iterations: 10
Their total (msec): 0
Our total (msec): 858
Their average (msec): 0
Our average (msec): 85
-----
compares 100000 Map.put() operations
Iterations: 10
Their total (msec): 6567
Our total (msec): 1365
Their average (msec): 656
Our average (msec): 136
-----
compares 20000 Set.contains() operations
Iterations: 10
Their total (msec): 277
Our total (msec): 209
Their average (msec): 27
Our average (msec): 20
-----
compares 100000 Map.get() operations
Iterations: 10
Their total (msec): 1002
Our total (msec): 853
Their average (msec): 100
Our average (msec): 85
-----
compares 100000 Set.add() operations
Iterations: 10
Their total (msec): 5411
Our total (msec): 1109
Their average (msec): 541
Our average (msec): 110
-----
compares ArrayList and TIntObjectHash get(i) over 100000 map keys
Iterations: 10
Their total (msec): 111
Our total (msec): 316
Their average (msec): 11
Our average (msec): 31
-----
```

```
compares HashMap and TLongObjectHashMap get() over 100000 map keys
Iterations: 10
Their total (msec): 1440
Our total (msec): 346
Their average (msec): 144
Our average (msec): 34
-----
compares HashMap and ArrayList retrieval over 100000 map keys
Iterations: 10
Their total (msec): 559
Our total (msec): 110
Their average (msec): 55
Our average (msec): 11
-----
compares ArrayList retrieval using size() over 100000 map keys
Iterations: 10
Their total (msec): 128
Our total (msec): 111
Their average (msec): 12
Our average (msec): 11
-----
compares ArrayList retrieval using new object and size() over 100000 map keys
Iterations: 10
Their total (msec): 122
Our total (msec): 112
Their average (msec): 12
Our average (msec): 11
-----
compares TObjectLongHashMap over 100000 map keys
Iterations: 10
Their total (msec): 1584
Our total (msec): 766
Their average (msec): 158
Our average (msec): 76
-----
compares ArrayList get() against Collection hasNext()/next() over 100000 map keys
Iterations: 10
Their total (msec): 248
Our total (msec): 110
Their average (msec): 24
Our average (msec): 11
-----
```

A.4 Linux IBM SDK memory benchmark results

```
-----
Compare size of Set implementation: 1,000 Integer objects measured in bytes
javasoft: 56221
trove: 25657
trove's collection requires 45% of the memory needed by javasoft's collection
-----
Compare size of LinkedList implementation: 1,000 TLinkableAdaptor objects measured in
bytes
javasoft: 44778
trove: 20754
trove's collection requires 46% of the memory needed by javasoft's collection
-----
Compare size of int/IntegerArrayList implementation: 1,000 ints measured in bytes
javasoft: 18444
trove: 3431
trove's collection requires 18% of the memory needed by javasoft's collection
-----
Compare size of Map implementation: 1,000 Integer->Integer mappings measured in bytes
javasoft: 54277
trove: 38400
trove's collection requires 70% of the memory needed by javasoft's collection
```

A.5 Linux Sun SDK memory benchmark results

```
-----  
Compare size of Set implementation: 1,000 Integer objects measured in bytes  
javasoft: 48262  
trove: 28856  
trove's collection requires 59% of the memory needed by javasoft's collection  
-----  
Compare size of LinkedList implementation: 1,000 TLinkableAdaptor objects measured in  
bytes  
javasoft: 40048  
trove: 16024  
trove's collection requires 40% of the memory needed by javasoft's collection  
-----  
Compare size of int/IntegerArrayList implementation: 1,000 ints measured in bytes  
javasoft: 20040  
trove: 4032  
trove's collection requires 20% of the memory needed by javasoft's collection  
-----  
Compare size of Map implementation: 1,000 Integer->Integer mappings measured in bytes  
javasoft: 48248  
trove: 41688  
trove's collection requires 86% of the memory needed by javasoft's collection
```

B Container test code

There are a few observations related to the testing conducted that do not have any impact on the analysis. They are however worth noting for those interested in adopting any of the measures used here.

B.1 Stateless session bean implementation

```
public class TesterBean extends java.lang.Object implements javax.ejb.SessionBean
{
    private SessionContext ctx;
    private Context naming = null;

    /**
     * Used to test the bean response time.
     *
     * @param counter      initial value
     *
     * @returns             <code>long</code> which should be counter
     */

    public final long serviceTest(long counter)
    {
        return counter;
    }

    /**
     * Used to test the lookup response.
     *
     * @returns             <code>long</code> which should be millisecond time
     */

    public final long serviceLookupTest()
    {
        long counter = 0;
        TesterLocalHome home;
        Date start = new Date();
        for (int i = 10000; i-- > 0; )
            try
            {
                home = (TesterLocalHome)naming.lookup("TesterLocal");
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        Date end = new Date();
        counter = end.getTime()-start.getTime();
        return counter;
    }

    /**
     * Used to test the local interface response.
     *
     * @returns             <code>long</code> which should be millisecond time
     */

    public final long serviceLocalTest()
    {
        long counter = 0;
        TesterLocalHome home = null;
        try
        {
            home = (TesterLocalHome)naming.lookup("TesterLocal");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        TesterLocal tester;
        Date start = new Date();
        for (int i = 10000; i-- > 0; )
```

```

        try
        {
            tester = (TesterLocal)home.create();
            counter = tester.serviceTest(counter);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        Date end = new Date();
        counter = end.getTime()-start.getTime();
        return counter;
    }

/**
 * Used to test the remote interface response.
 *
 * @returns      <code>long</code> which should be millisecond time
 */

public final long serviceRemoteTest()
{
    long counter = 0;
    TesterHome home = null;
    try
    {
        Object reference = naming.lookup("Tester");
        home = (TesterHome)PortableRemoteObject.narrow(reference,
            TesterHome.class);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    Tester tester;
    Date start = new Date();
    for (int i = 10000; i-- > 0; )
        try
        {
            tester = (Tester)home.create();
            counter = tester.serviceTest(counter);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    Date end = new Date();
    counter = end.getTime()-start.getTime();
    return counter;
}

/**
 * Standard EJB setSessionContext.  Contains no code.
 *
 * @param context      environmental context if necessary
 */

public void setSessionContext(javax.ejb.SessionContext context)
    throws javax.ejb.EJBException, java.rmi.RemoteException
{
}

/**
 * Standard EJB ejbCreate.  Required by home interface.  Contains no code.
 */

public void ejbCreate() throws javax.ejb.CreateException
{
    try
    {
        {
            naming = new InitialContext();
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}

```

```

}

/**
 * Standard EJB ejbRemove.  Contains no code.
 */

public void ejbRemove() throws javax.ejb.EJBException, java.rmi.RemoteException
{
}

/**
 * Standard EJB ejbActivate.  Contains no code.
 */

public void ejbActivate() throws javax.ejb.EJBException, java.rmi.RemoteException
{
}

/**
 * Standard EJB ejbPassivate.  Contains no code.
 */

public void ejbPassivate() throws javax.ejb.EJBException, java.rmi.RemoteException
{
}
}

```

B.2 Test load client code

The code that starts the tests is fairly raw but gets the job done. It is provided here for clarity on the test operation.

```

public class Main extends java.lang.Object
{
    /**
     * Used to test the lookup response.
     *
     * @returns      <code>long</code> which should be millisecond time
     */

    class ServiceRemoteLookupTest extends Thread
    {
        private int id = 0;
        private Properties jndiProps = null;

        public ServiceRemoteLookupTest(int id, Properties jndiProps)
        {
            this.id = id;
            this.jndiProps = jndiProps;
        }

        public void run()
        {
            TesterHome home;
            Object reference;
            Context naming = null;
            long counter = 0;
            try
            {
                naming = new InitialContext(jndiProps);
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
            Date start = new Date();
            for (int i = 10000; i-- > 0; )
                try
                {
                    reference = naming.lookup("Tester");
                    home = (TesterHome)PortableRemoteObject.narrow(reference,
                        TesterHome.class);
                }
                catch(Exception e)

```

```

        {
            e.printStackTrace();
        }
        Date end = new Date();
        counter = end.getTime()-start.getTime();
        System.out.println("Remote lookup " + id + ": " + counter + " ms");
    }
}

/**
 * Used to test the remote interface response.
 *
 * @returns      <code>long</code> which should be millisecond time
 */

class ServiceRemoteTest extends Thread
{
    private int id = 0;
    private Properties jndiProps = null;

    public ServiceRemoteTest(int id, Properties jndiProps)
    {
        this.id = id;
        this.jndiProps = jndiProps;
    }

    public void run()
    {
        TesterHome home = null;
        long counter = 0;
        try
        {
            Context naming = new InitialContext(jndiProps);
            Object reference = naming.lookup("Tester");
            home = (TesterHome)PortableRemoteObject.narrow(reference,
                TesterHome.class);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        Tester tester;
        Date start = new Date();
        for (int i = 10000; i-- > 0; )
            try
            {
                tester = (Tester)home.create();
                counter = tester.serviceTest(counter);
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        Date end = new Date();
        counter = end.getTime()-start.getTime();
        System.out.println("Remote " + id + ": " + counter + " ms");
    }
}

/**
 * Used to test the local lookup response.
 *
 * @returns      <code>long</code> which should be millisecond time
 */

class ServiceLocalLookupTest extends Thread
{
    private int id = 0;
    private Properties jndiProps = null;

    public ServiceLocalLookupTest(int id, Properties jndiProps)
    {
        this.id = id;
        this.jndiProps = jndiProps;
    }
}

```

```

public void run()
{
    TesterHome home = null;
    long counter = 0;
    try
    {
        Context naming = new InitialContext(jndiProps);
        Object reference = naming.lookup("Tester");
        home = (TesterHome)PortableRemoteObject.narrow(reference,
            TesterHome.class);
    }
    catch(Exception e)
    {
        e.printStackTrace();
        return;
    }
    Tester tester;
    Date start = new Date();
    try
    {
        tester = (Tester)home.create();
        counter = tester.serviceLookupTest();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    System.out.println("Local Lookup " + id + ": " + counter + " ms");
}
}

/**
 * Used to test the local interface response.
 *
 * @returns      <code>long</code> which should be millisecond time
 */

class ServiceLocalTest extends Thread
{
    private int id = 0;
    private Properties jndiProps = null;

    public ServiceLocalTest(int id, Properties jndiProps)
    {
        this.id = id;
        this.jndiProps = jndiProps;
    }

    public void run()
    {
        TesterHome home = null;
        long counter = 0;
        try
        {
            Context naming = new InitialContext(jndiProps);
            Object reference = naming.lookup("Tester");
            home = (TesterHome)PortableRemoteObject.narrow(reference,
                TesterHome.class);
        }
        catch(Exception e)
        {
            e.printStackTrace();
            return;
        }
        Tester tester;
        Date start = new Date();
        try
        {
            tester = (Tester)home.create();
            counter = tester.serviceLocalTest();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

```

        System.out.println("Local interface " + id + ": " + counter + " ms");
    }
}

/**
 * Used to test the remote interface response.
 *
 * @returns      <code>long</code> which should be millisecond time
 */

class ServiceLocalRemoteTest extends Thread
{
    private int id = 0;
    private Properties jndiProps = null;

    public ServiceLocalRemoteTest(int id, Properties jndiProps)
    {
        this.id = id;
        this.jndiProps = jndiProps;
    }

    public void run()
    {
        TesterHome home = null;
        long counter = 0;
        try
        {
            Context naming = new InitialContext(jndiProps);
            Object reference = naming.lookup("Tester");
            home = (TesterHome)PortableRemoteObject.narrow(reference,
                TesterHome.class);
        }
        catch(Exception e)
        {
            e.printStackTrace();
            return;
        }
        Tester tester;
        Date start = new Date();
        try
        {
            {
                tester = (Tester)home.create();
                counter = tester.serviceRemoteTest();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
        System.out.println("Local remote interface " + id + ": " + counter
            + " ms");
    }
}

public void localLookup(int threads, Properties jndiProps)
{
    for (int i = 0; i < threads; i++)
        new ServiceLocalLookupTest(i, jndiProps).start();
}

public void localTest(int threads, Properties jndiProps)
{
    for (int i = 0; i < threads; i++)
        new ServiceLocalTest(i, jndiProps).start();
}

public void localRemote(int threads, Properties jndiProps)
{
    for (int i = 0; i < threads; i++)
        new ServiceLocalRemoteTest(i, jndiProps).start();
}

public void remoteLookup(int threads, Properties jndiProps)
{
    for (int i = 0; i < threads; i++)
        new ServiceRemoteLookupTest(i, jndiProps).start();
}
}

```

```

public void remoteTest(int threads, Properties jndiProps)
{
    for (int i = 0; i < threads; i++)
        new ServiceRemoteTest(i, jndiProps).start();
}

/**
 * @param args the command line arguments
 */
public static void main(String[] args)
{
    String NCFACTORY = "org.jnp.interfaces.NamingContextFactory";
    String NPURL = "jnp://localhost:1099";
    String NPKGS = "org.jboss.naming.org.jnp.interfaces";
    String JNDIFACTORY = "java.naming.factory.initial";
    String JNDIURL = "java.naming.provider.url";
    String JNDIPKGS = "java.naming.factory.url.pkgs";
    Properties jndiProps = new Properties();
    Properties props = System.getProperties();
    boolean embedded = props.containsKey(JNDIFACTORY) &&
        props.containsKey(JNDIPKGS);

    if (!embedded)
    {
        jndiProps.setProperty(JNDIPKGS, NPKGS);
        jndiProps.setProperty(JNDIFACTORY, NCFACTORY);
    }
    if (!props.containsKey(JNDIURL) && !embedded)
        jndiProps.setProperty(JNDIURL, NPURL);
    if (jndiProps.containsKey(JNDIURL))
    {
        jndiProps.setProperty("jnp.socketFactory",
            "org.jnp.interfaces.TimedSocketFactory");
        jndiProps.setProperty("jnp.timeout", "0");
        jndiProps.setProperty("jnp.sotimeout", "0");
    }
    Main test = new Main();
    if (args.length < 2)
    {
        System.err.println("Expected selection information.");
        System.exit(1);
    }
    try
    {
        int i = Integer.parseInt(args[0]);
        int j = Integer.parseInt(args[1]);
        switch(i)
        {
            case 1:
                test.localLookup(j, jndiProps);
                break;
            case 2:
                test.localTest(j, jndiProps);
                break;
            case 3:
                test.localRemote(j, jndiProps);
                break;
            case 4:
                test.remoteLookup(j, jndiProps);
                break;
            case 5:
                test.remoteTest(j, jndiProps);
                break;
            default:
                break;
        }
    }
    catch(Exception e)
    {
        System.err.println("Arguments expected to be integers.");
        System.exit(2);
    }
}
}

```