Understanding JSF 2 and Wicket : Performance Comparison

Leonardo A. Uribe P.
PMC member Apache MyFaces Project
lu4242@apache.org
May 2012

Abstract — *This article highlights last year's (2011-2012) performance enhancements done within Apache MyFaces JSF Implementation, through an in-deep comparison between JSF and Wicket. The comparison is done looking from different aspects (speed, memory usage, session size), to give a better understanding about how each framework works under different conditions. At the end, choose a web framework requires to balance not only performance but other different considerations.*

Keywords- *Java; Java Server Faces; JSF; Wicket; Java Web Frameworks; Performance Comparison*

I. Introduction

Performance is important in a web framework, even though it is certainly not the only consideration. This document highlights last year's performance enhancements done within the Apache MyFaces JSF Implementation (aka MyFaces Core).

To compare JSF against other web frameworks, the reference application already provided in a blog about seam JSF vs Wicket was used. [1] Fortunately, the wicket guys have been so kind to updated the original application running with 1.4.x to version 1.5.x, so we'll compare JSF against two Wicket versions (1.4.20 and 1.5.5). A comparison against Tapestry might happen later. In the end, the intention is to look at how JSF works from a performance perspective, so Wicket will be taken only as a reference point.

Checking the code provided in that blog and the related data, some rather serious problems were discovered:

- Only one user (demo) is shared for all http requests. This causes a contention problem inside hibernate. The solution is to use a different user per thread.
- The same HttpCookieManager is used for all threads, so only one session is created. This causes another contention problem. The solution is to use a HttpCookieManager per thread.
- The test data omits important information about the experiment itself, giving the wrong impression. For example, there is no mention about the number of processors used, the JVM version and its configuration. Additionally, it uses the average value for the comparison, which could be distorted easily if a request takes a long time to be processed.
- Once a booking is created, it is not canceled, so the view that list all bookings has more and more rows, depending on the previous requests.

Anyway, the code example is good enough to reuse it, while obviously fixing the detected problems. The updated code, configuration used, detailed documentation and experimental data can be found in the internet (github).[2]

A good benchmark should consider different aspects. So, for this comparison, the following aspects will be checked:

1. http://ptrthomas.wordpress.com/2009/01/14/seam-jsf-vs-wicket-performance-comparison/
http://perfbench.googlecode.com/svn/trunk/perfbench/
2 https://github.com/lu4242/performance-comparison-java-web-frameworks

- Speed benchmark: Check the speed of the code under different situations.
- Memory Consumption : Involves checking how much memory does the application need to allocate for a specific task, and how the application behaves under low memory conditions.
- Session Size : Check how much memory is used to store the session.

It is quite obvious why these three aspects should be considered:

- If the framework is fast it will process more requests with less CPU consumption.
- If the framework requires less memory, less time will be used in garbage collection and the framework will perform better under stress conditions.
- If it uses less memory as session storage, it will be able to serve more users with the same memory size, and if the session is stored somewhere (database, file, memory cache...), less time will be used in processing that information (usually serialization/deserialization and dealing with I/O operations).

A. About the test project

The test project is a simple hotel booking application example provided by Seam 2.2.x and ported to different web frameworks. In this case, the seam-jpa example was taken, and the code was rewritten to use JSF 2.0 and JPA in the same way the wicket-jpa example does. In this way, the results are comparable, because the business logic and data access layer work more or less the same.

The operations considered in the test are:

- get login
- post login
- ajax post search
- get view hotel
- post book hotel
- ajax post cc number
- ajax post cc name
- post booking details
- post confirm booking
- get cancel booking
- logout

It is quite simple to run the application, just build the code, take the war and deploy it on a server. An in-memory database is created with the data, so no additional configuration is necessary.

B. Initial considerations

The experiment is based on the following assumptions:

- First of all, the intention of these tests is gather some benchmark data- this is not a load test or a stress test. Hence the experiments are designed to provide valuable data that later can be useful to understand how JSF works.

- For the comparison, the MyFaces 2.1.7 and Mojarra 2.1.7 JSF implementations are used. OnlyJSR-303 Bean Validation and no other jsf tags different to the ones provided by the JSF standard are used. In Wicket, hibernate-validator 3.1.0.GA will be used because it just fits better, and there is not a standard or good enough solution to use JSR-303 with Wicket. With JSF, hibernate-validator 4.0.2.GA is used.

- The Facelets algorithm works in two steps. First, it reads some xhtml definitions and build an Abstract Syntax Tree (AST), that will be later used to build multiple views. Second, it applies the AST to build a JSF Component tree. The first step is done only the first time the view is created, so before test start it is important to load the views running the test just once. It is similar to JSP, but in that case a jsp page is compiled the first time it is called. With JSP, there is a setting to do this on startup, this setting is not available within Facelets. Hence, before running the test, a warmup cycle should be executed to exclude this effect.

- Additionally it is important to consider the JVM warmup time (due to optimization ocurring within the JVM). In this case, the effect seen is the server takes less and less time to handle a specific request, until the request processing time reaches a lower limit, usually related to the JVM version. If a speed test is done, it is better to take the data after the JVM has enough warmup time.

- To ensure comparable data between MyFaces and Mojarra, encryption has been disabled in MyFaces, because Mojarra does not use it, but data has been provided to compare MyFaces with and without default encryption when it is necessary. Server side state saving is enabled and some flags that improve performance in MyFaces have been enabled like:

```
<context-param>
  <param-name>org.apache.myfaces.CACHE_EL_EXPRESSIONS</param-name>
  <param-value>always</param-value>
</context-param>

<context-param>
  <param-name>org.apache.myfaces.CHECK_ID_PRODUCTION_MODE</param-name>
  <param-value>false</param-value>
</context-param>

<context-param>
  <param-name>org.apache.myfaces.SUPPORT_JSP_AND_FACES_EL</param-name>
  <param-value>false</param-value>
</context-param>

<context-param>
  <param-name>org.apache.myfaces.VIEW_UNIQUE_IDS_CACHE_ENABLED</param-name>
  <param-value>true</param-value>
</context-param>

<context-param>
  <param-name>org.apache.myfaces.SAVE_STATE_WITH_VISIT_TREE_ON_PSS</param-name>
  <param-value>false</param-value>
</context-param>
```

- JSF uses EL heavily and the performance of the used EL affects performance of JSF mainly in the render response phase.[3] There are more implementations with different performance, but their comparison is out of scope of this document. For the tests, the

3 When the view is built, it is necessary to create EL expressions. Later when the view is rendered, to get information from the model it is necessary to evaluate such expressions.

tomcat EL implementation was used.[4]

- For Wicket 1.4.20 test and Tomcat, jmeter doesn't handle "post confirm booking" redirect, so the POST and GET were split into two requests, but these requests count as one and this does not affect the tests.

- Logging information has been disabled.

- Apache Tomcat 7.0.26 is used as a web server. To improve accuracy, auto deploy has been disabled (<Host autoDeploy=false>) and AccessLogValve has been removed.

- In JMeter, for warmup and stress tests Httpclient4 is used as Http request implementation, because it closes the sockets and prevents errors in that part, but for the speed tests the Java mode is used, because it causes less overhead over each request.

- Wicket behaves different according to its page storage mode. Disk storage (default) and Http session storage are considered in the comparison.

- There is one computer that runs Tomcat and there is another computer that runs JMeter. When Netbeans Profiler or YourKit Profiler are used, they run on the same machine that runs Tomcat.

II. Speed Benchmark

A. Part I: Code speed with minimal load.

In this first part, the intention is take a look at how each framework perform under minimum load. The data gathered here are useful to identify bottlenecks in code speed with minimal concurrency effect. Based on the previous considerations, the following experiment was done.

CONFIGURATION	**WARMUP**	**EXPERIMENT**
Processor: AMD Phenom x4 Speed: 2300 MHz Server: Apache Tomcat 7.0.26 JVM version : oracle jdk1.6.0_30 JVM Options : -Xms128m -Xmx128m -server	loop.count: 1500 thread.count: 5 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 10 thread.deviation: 5 rampup time: 1s Http Request Client : Httpclient4	loop.count: 100 thread.count: 5 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 0 thread.deviation:0 rampup time: 1s Http Request Client : Java

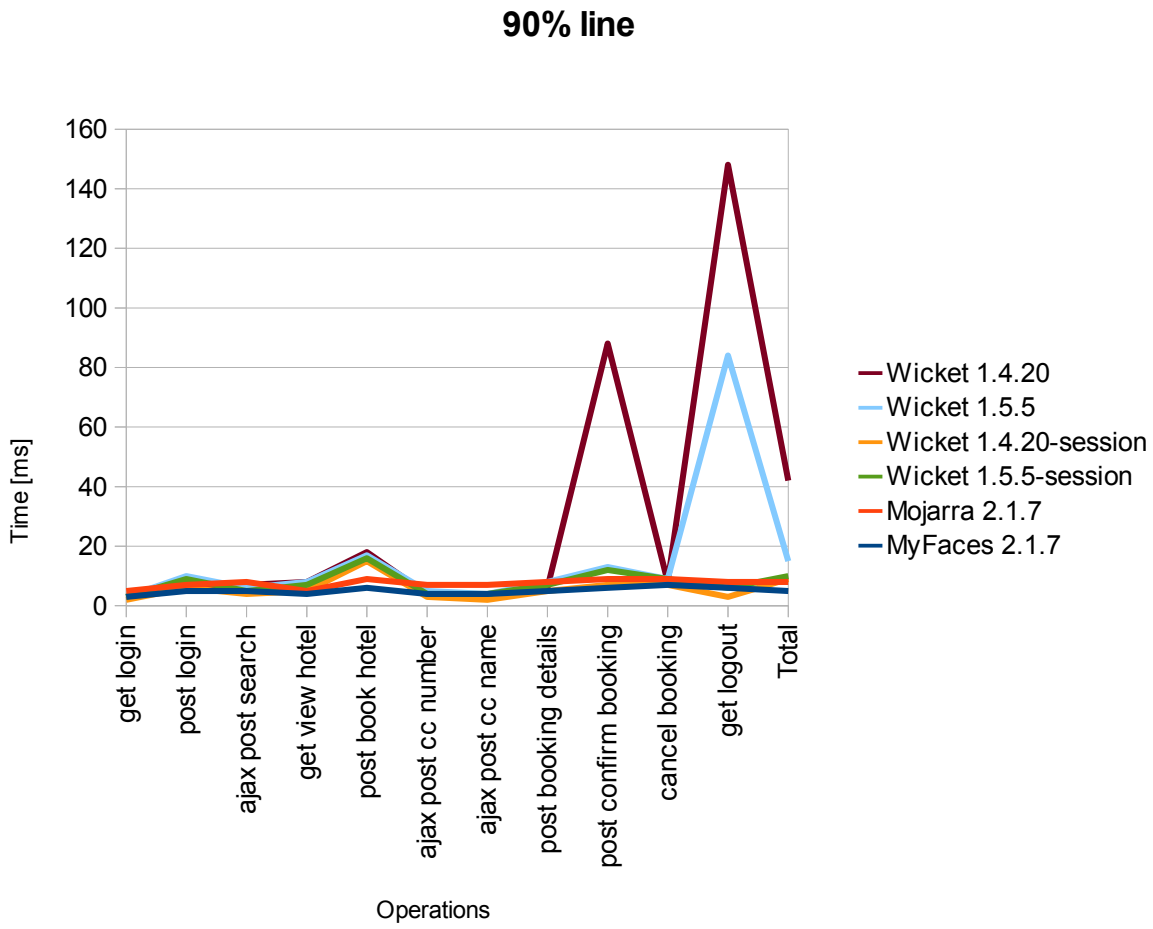
This is the median and 90% line:

4 For tomcat 7.0.26, it is known this issue https://issues.apache.org/bugzilla/show_bug.cgi?id=52998 causes an overhead when creating EL expressions.

	MyFaces 2.1.7		Mojarra 2.1.7	
Operation	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]
get login	2	3	3	3
post login	3	5	5	5
ajax post search	4	5	5	5
get view hotel	3	4	3	4
post book hotel	4	6	7	6
ajax post cc number	3	4	5	4
ajax post cc name	3	4	5	4
post booking details	4	5	6	5
post confirm booking	5	6	7	7
cancel booking	5	7	7	6
get logout	4	6	5	6
TOTAL	4	5	5	6

	Wicket 1.4.20		Wicket 1.4.20 session storage		Wicket 1.5.5		Wicket 1.5.5 session storage	
Operation	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]
get login	2	3	2	2	2	3	2	3
post login	5	7	4	6	7	10	7	9
ajax post search	4	7	3	4	4	6	4	5
get view hotel	4	8	4	5	4	8	4	7
post book hotel	12	18	12	15	12	17	12	16
ajax post cc number	2	4	2	3	3	5	3	4
ajax post cc name	2	3	2	2	3	4	3	4
post booking details	4	6	4	5	5	8	5	7
post confirm booking	34	88	5	7	8	13	8	12
cancel booking	6	8	5	7	7	9	7	9
get logout	69	148	2	3	4	84	4	6
TOTAL	5	42	3	9	4	15	4	10

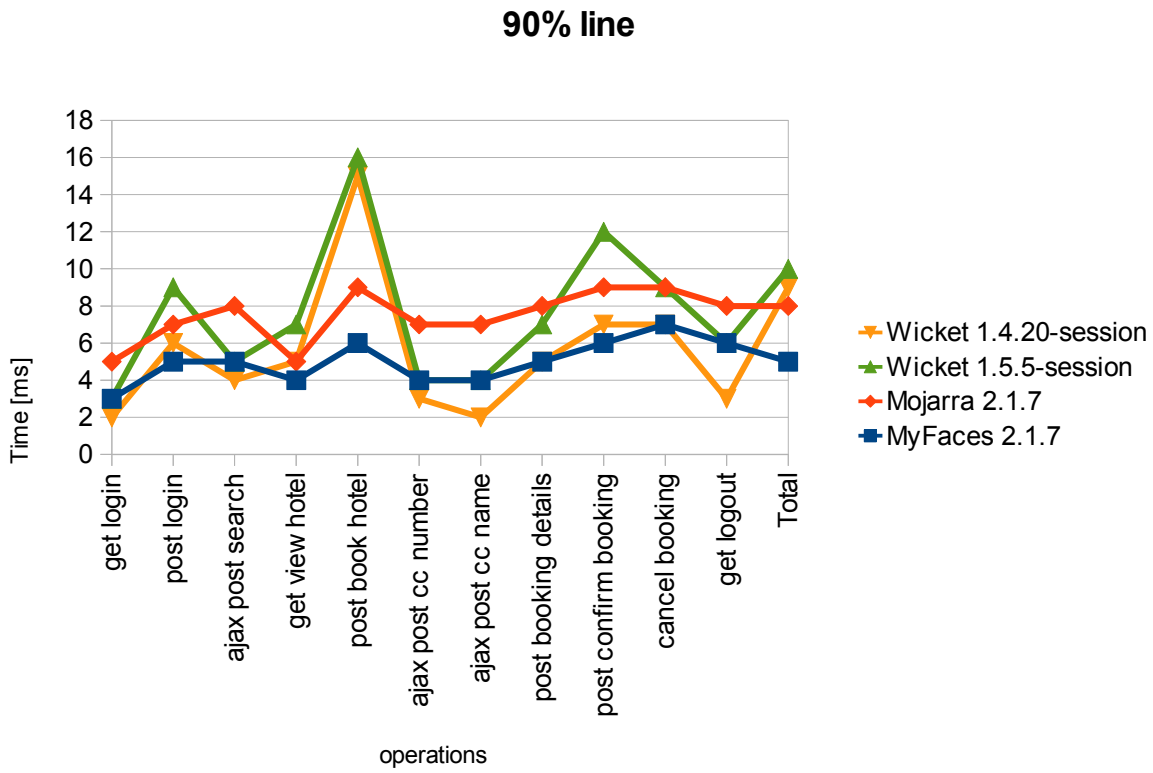
Note Wicket behaves different when disk page storage is used (default) and when http session page storage is used. The data gathered consider this effect and you can see the effect of the additional I/O operations. To compare against JSF, it is better to use session storage. See the graph below:



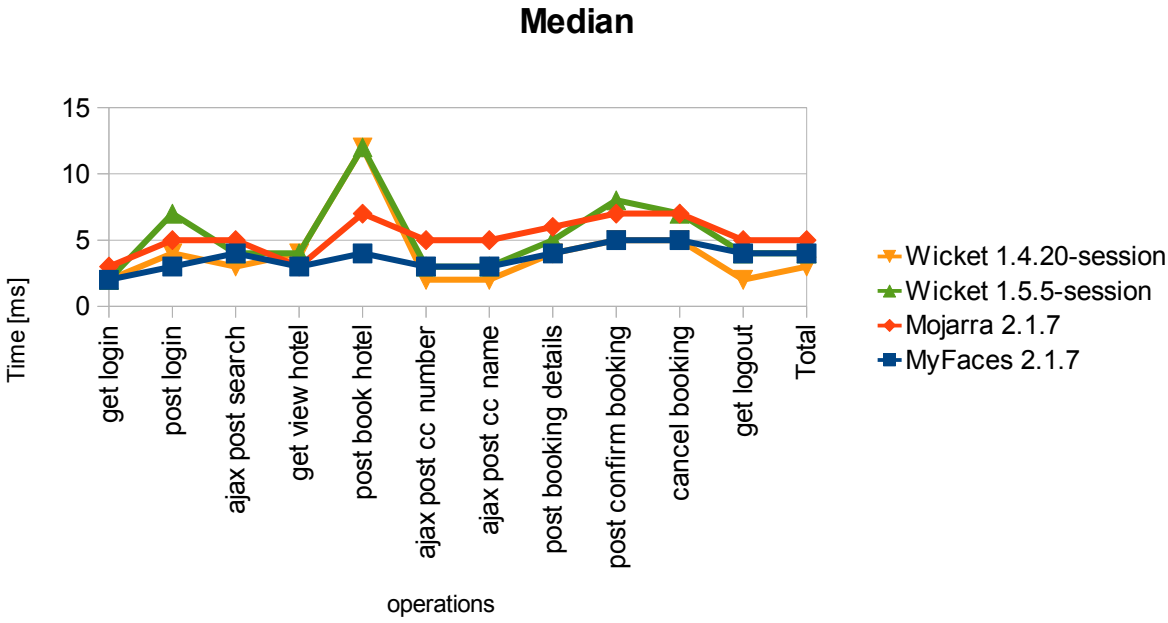
When disk storage is used and the session is invalidated (get logout) it is necessary to remove some data from disk. Checking this effect for the application, in "post confirm booking", some pages are removed from the storage, so you can see a similar effect in Wicket 1.4.20. Note the effect in Wicket 1.5.5 is not present in "post confirm booking", but it is present in the logout. At the end, using disk storage is a compromise between CPU and session size: at the cost of some additional I/O and serialization/deserialization of the page map, session size per user is reduced.

JSF uses an efficient Partial State Saving algorithm to store just the necessary information on the session and does not need to deal with these I/O operations. Remember performance is a balance, and the interesting idea to keep in mind here is that Wicket prefers to sacrify CPU speed to reduce session size.

Now let's see a comparison graph between Wicket and JSF



The first thing you see is Wicket GET requests sometimes are very close to MyFaces 2.1.7, sometimes are faster, but some POST request are slower. Note that the parallelism effect is minimal in this comparison. The median shows a similar effect.



Note wicket 1.5.5 is slower than 1.4.20, and that MyFaces 2.1.7 is faster than Mojarra 2.1.7, but with minimal concurrency MyFaces 2.1.7 is very close to Wicket 1.4.20.

B. Part 2: Code speed with load.

Now let's see what happen if we run the tests, but this time using more threads, keeping enough memory resources. As explained before, the relevant data in this case is the median and specially the 90% line, because with those values it is possible to see how multiple threads works, "how much contention overhead" a web framework has, and the results will be more closer to the reality. These are the experiment parameters:

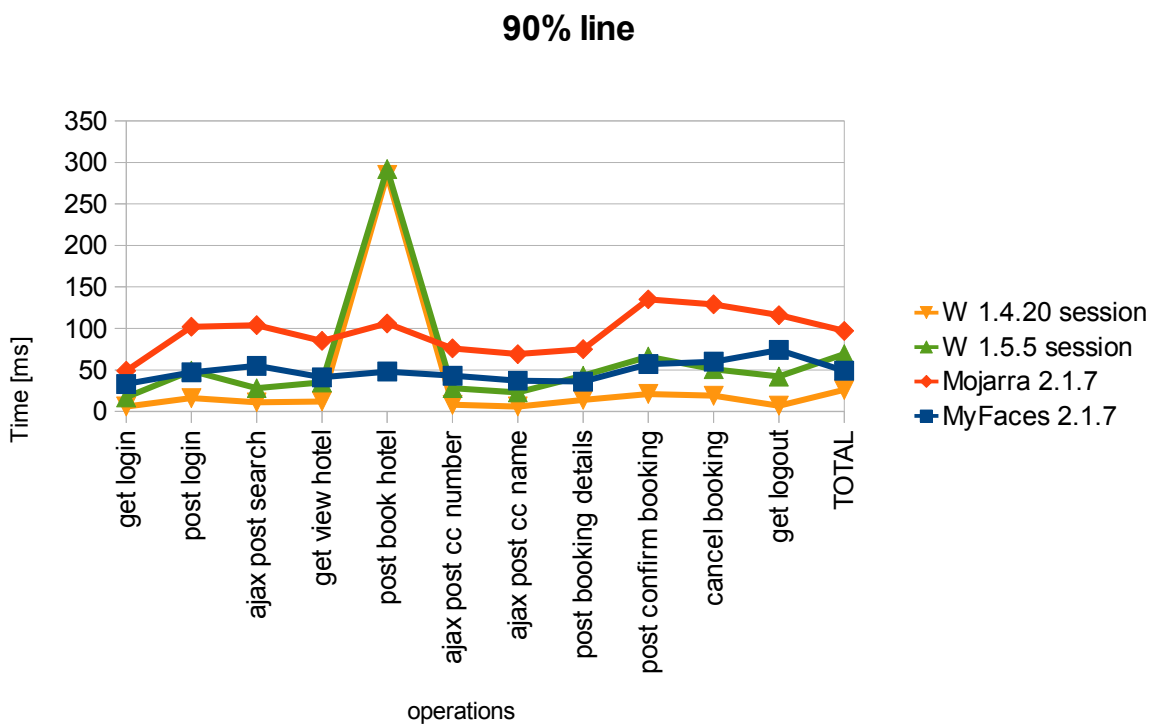
CONFIGURATION	WARMUP	EXPERIMENT
Processor: AMD Phenom x4 Speed: 2300 MHz Server: Apache Tomcat 7.0.26 JVM version : oracle jdk1.6.0_30 JVM Options : -Xms128m -Xmx128m -server	loop.count: 200 thread.count: 40 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 20 thread.deviation:10 rampup time: 1s Http Request Client : Httpclient4	loop.count: 100 thread.count: 40 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 0 thread.deviation:0 rampup time: 5s Http Request Client : Java

Note the rampup time on the experiment was increased to 5 seconds , because there are more threads to create. These are the results:

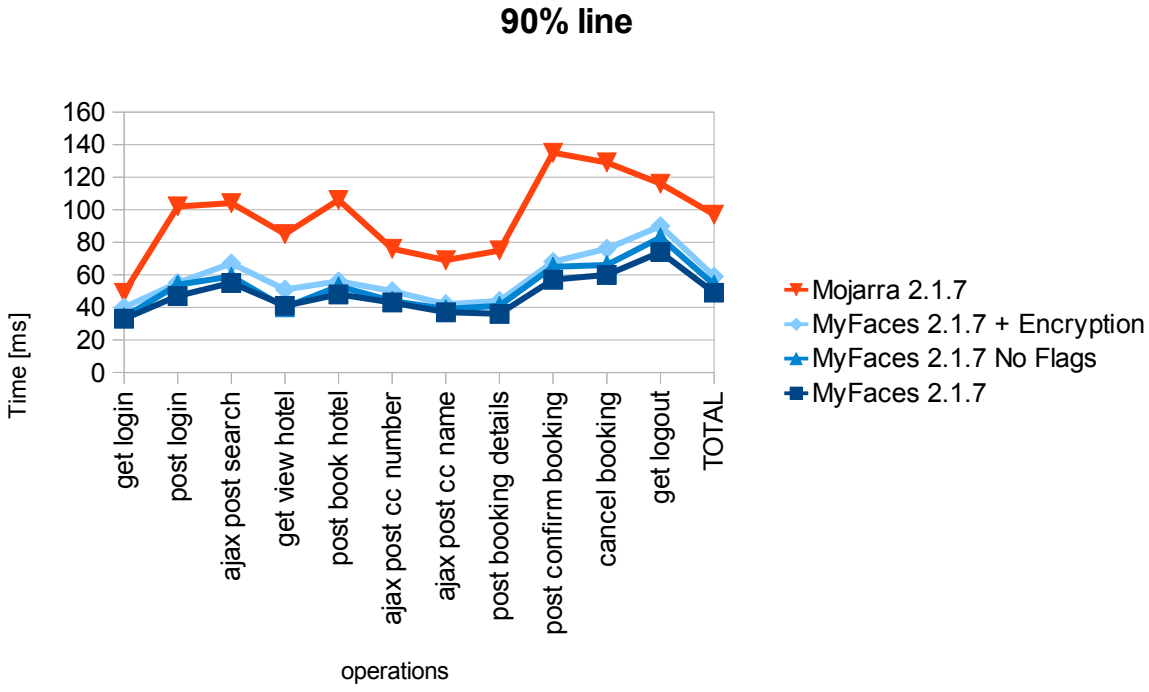
	MyFaces 2.1.1		MyFaces 2.1.6		MyFaces 2.1.7		MyFaces 2.1.7 + Encryption		MyFaces 2.1.7 No Flags	
Operation	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]
get login	34	83	11	59	5	33	5	40	5	34
post login	102	200	41	121	6	47	8	55	7	54
ajax post search	58	111	19	72	7	55	8	67	8	59
get view hotel	71	134	21	85	6	41	6	51	6	40
post book hotel	60	111	19	72	7	48	8	56	8	53
ajax post cc number	36	87	14	62	5	43	6	50	6	44
ajax post cc name	58	112	13	61	5	37	6	42	6	39
post booking details	164	338	15	64	7	36	7	44	7	41
post confirm booking	57	121	28	93	9	57	10	68	9	65
cancel booking	43	92	20	76	9	60	10	76	9	66
get logout	78	142	29	87	12	74	14	90	13	83
TOTAL	62	149	20	79	7	49	8	59	7	54

	Wicket 1.4.20		Wicket 1.4.20 session storage		Wicket 1.5.5		Wicket 1.5.5 session storage		Mojarra 2.1.7	
Operation	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]	Median [ms]	90% line [ms]
get login	2	3	2	6	2	5	3	17	7	49
post login	5	7	6	16	15	161	13	49	11	102
ajax post search	3	5	4	11	7	43	6	28	12	104
get view hotel	4	7	5	12	9	72	8	35	8	85
post book hotel	12	20	183	285	45	143	162	292	12	106
ajax post cc number	2	4	3	8	6	43	5	28	11	76
ajax post cc name	2	4	2	6	6	43	5	23	10	69
post booking details	4	7	5	14	11	76	10	43	10	75
post confirm booking	390	473	7	21	17	88	14	66	14	135
cancel booking	6	9	7	19	12	46	13	51	14	129
get logout	719	833	2	7	242	475	8	42	21	116
TOTAL	5	434	4	26	10	106	9	69	11	97

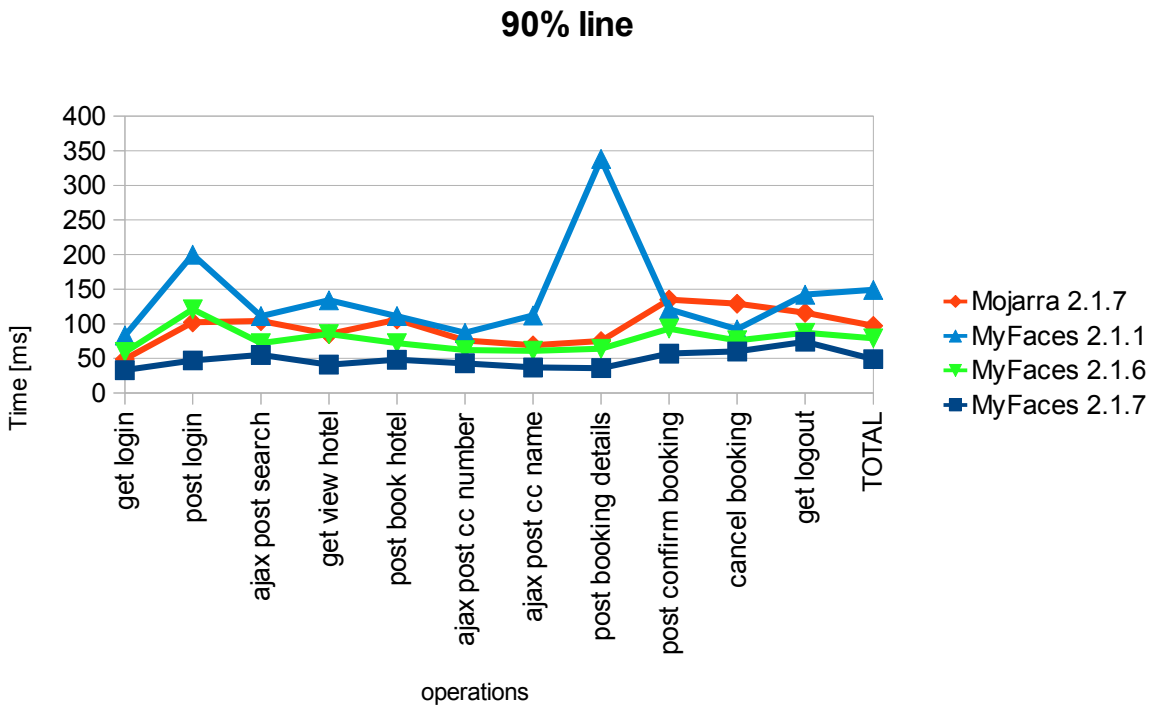
Let's see the 90% line of wicket with session storage and JSF:



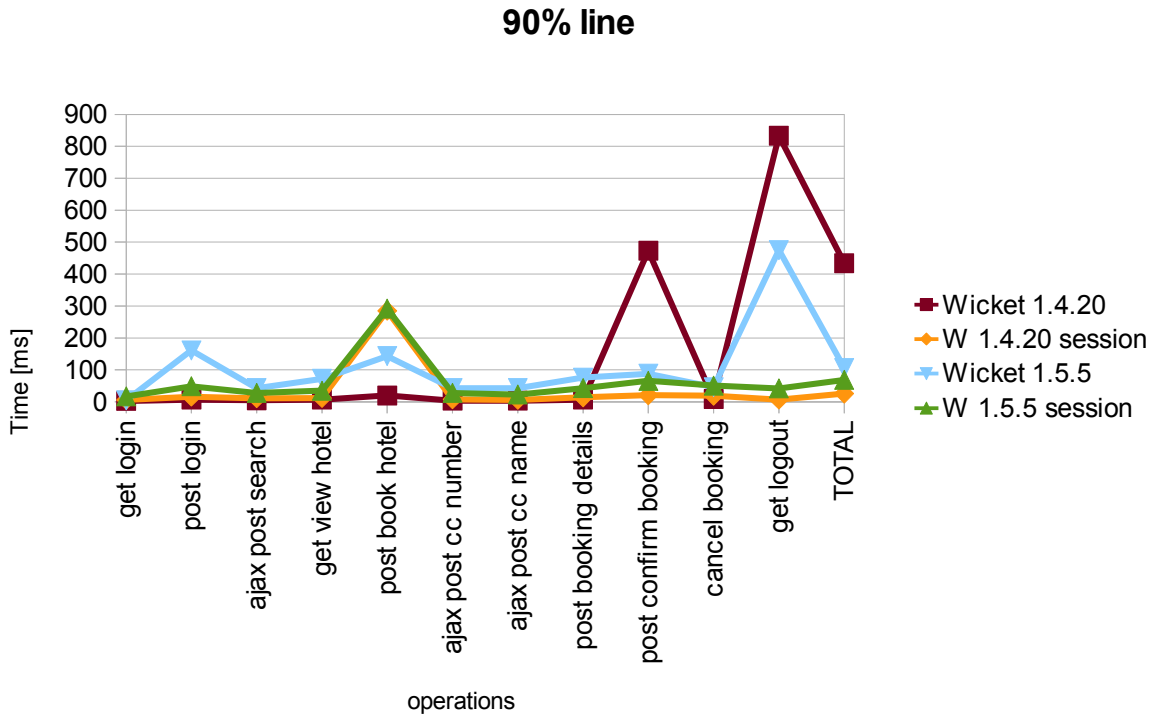
Here Wicket 1.4.20 is the fastest one. Comparing these data with the ones in the first part, you can conclude Wicket speed is more related to the concurrency effect. Note how slow the "post book hotel" is compared with JSF, and how close MyFaces 2.1.7 is to Wicket 1.5.5.



Here you can see that the cost of encrypting the state in MyFaces is reasonable, compared with Mojarra, and that the flags do have some effect on the speed. Now let's see the same information for previous versions of MyFaces:



Here you can see the steady improvement since 2.1.1, especially in 2.1.6 and 2.1.7. This wouldn't be possible without a lot of feedback received from the MyFaces community.



Definitively, the logout case (or invalidate/clear session) is the most detrimental for Wicket

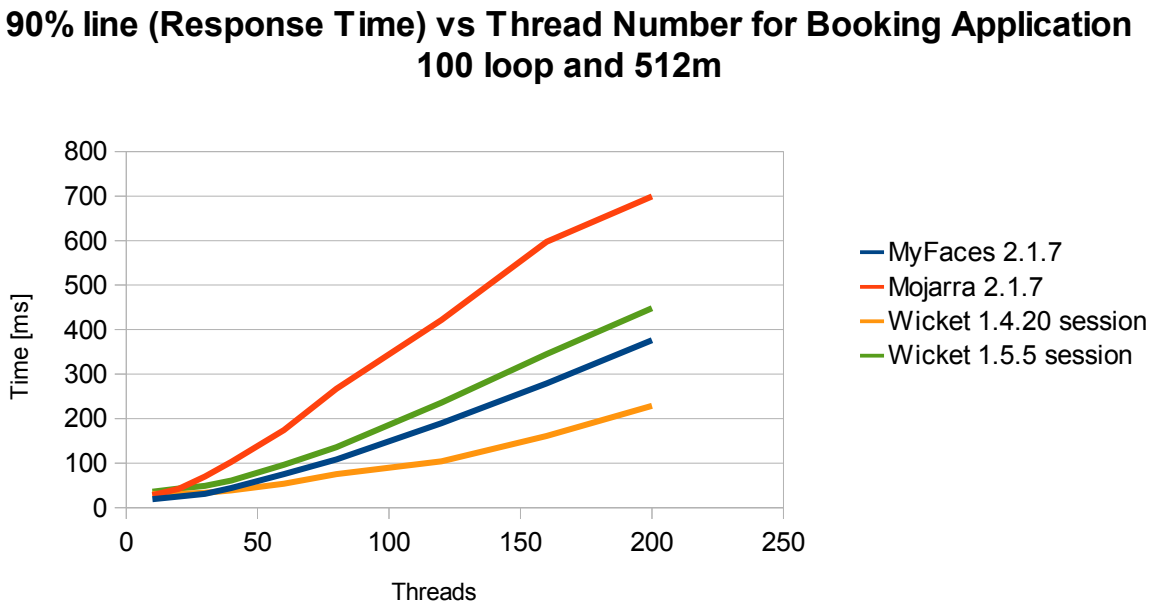
performance. You can see how expensive the disk storage is for Wicket. The associated cost will depend of the underlying hardware environment, but definitively it is something that should be kept into account.

C. Part 3: Code speed with increasing load

In order of validate the results gathered in part 2, it is necessary to check the behavior when the load is increasing, varying the number of threads. Some changes were done like increasing the memory to 512m, instead of using Java as http request client Httpclient4 is used, and we provide a longer warmup time. Here is the experiment data:

CONFIGURATION	WARMUP	EXPERIMENT
Processor: AMD Phenom x4 Speed: 2300 MHz Server: Apache Tomcat 7.0.26 JVM version : oracle jdk1.6.0_16 for linux JVM Options : -Xms512m -Xmx512m -server	loop.count: 200 thread.count: 100 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 20 thread.deviation:10 rampup time: 20s Http Request Client : Httpclient4	loop.count: 100 thread.count: 10, 20, 30, 40, 60, 80, 120, 160, 200 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 0 thread.deviation:0 rampup time: 1, 2, 3, 4, 6, 8, 12, 16, 20s Http Request Client : Httpclient4

The changes done do not lead to any significant difference. In this case, Wicket 1.4 and 1.5 with disk storage (default) have been omitted, due to the effects described earlier. Here is the resulting graph for the 90% line:



The graphs shows that the conclusions in part 2 are still valid even if the number of threads are increased.

As you can see, the results are not very conclusive. Since there is no uniform behavior between JSF and Wicket, the concrete benchmark or in other words the application used at the end will decide who is the winner. Comparing MyFaces and Mojarra in the graph, you can see MyFaces 2.1.7 has become the fastest by a wide margin (more than 40%!). Comparing Wicket 1.4.20 and 1.5.5 you can see 1.4.20 is faster. In this case, a better programming model in 1.5.5 has costed some speed. At the

end, each framework has different trade-offs. It is necessary to look in deep how each framework uses available memory and how session size grows to get a better idea.

III. Memory

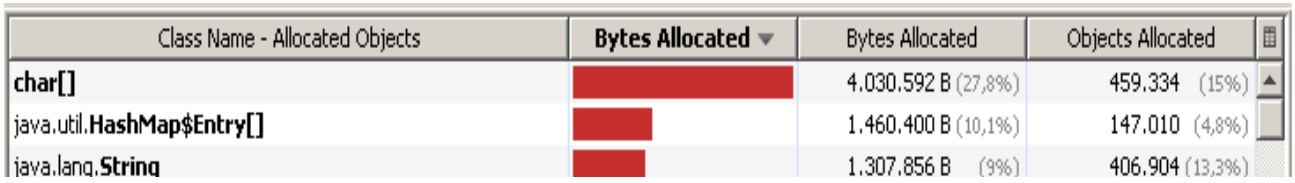
As was mentioned before, another important factor isto take into account how the available memory is used. In a few words, from a performance perspective it is important to keep memory allocation to a minimum because:

- Allocating memory takes time.
- Each call to the garbage collector (GC) is expensive and at the end slows down the web server.

To see how memory is used, the following experiment is run with a profiler attached to see how many objects are created. Here are the results:

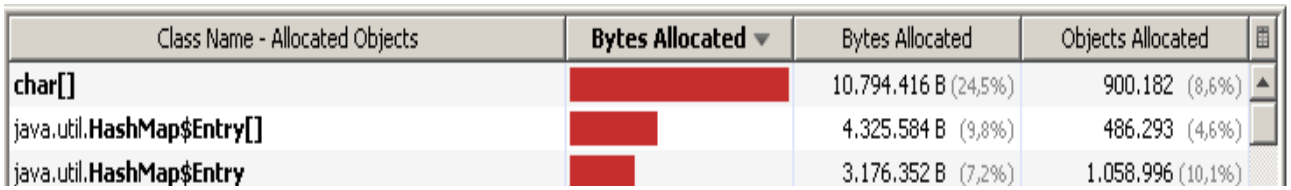
CONFIGURATION	**WARMUP**	**EXPERIMENT**
Processor: AMD Phenom x4 Speed: 2300 MHz Server: Apache Tomcat 7.0.26 JVM version : oracle jdk1.6.0_30 JVM Options : -Xms128m -Xmx128 -server	loop.count: 20 thread.count: 5 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 0 thread.deviation: 0 rampup time: 1s	loop.count: 20 thread.count: 5 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 0 thread.deviation:0 rampup time: 1s

MyFaces 2.1.7



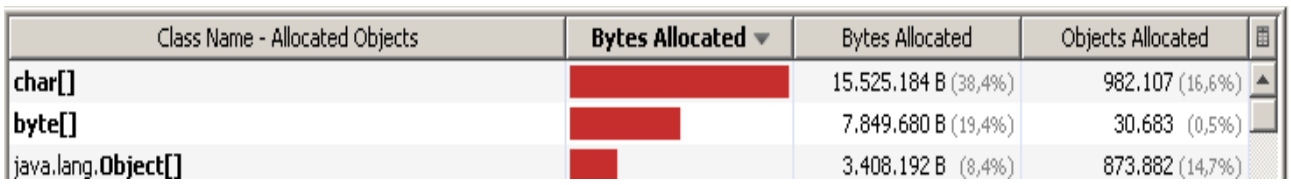
Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		4.030.592 B (27,8%)	459.334 (15%)
java.util.HashMap$Entry[]		1.460.400 B (10,1%)	147.010 (4,8%)
java.lang.String		1.307.856 B (9%)	406.904 (13,3%)

Mojarra 2.1.7



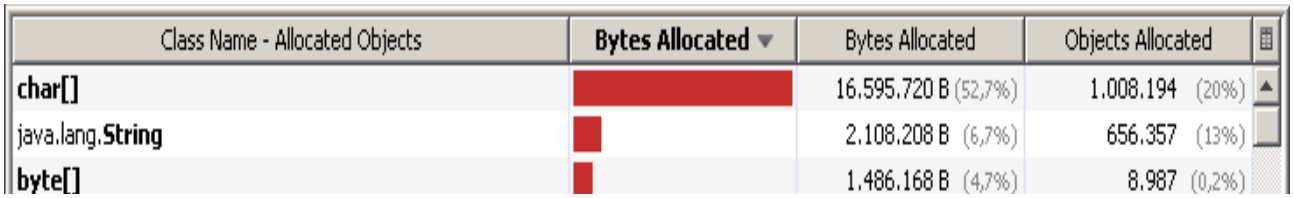
Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		10.794.416 B (24,5%)	900.182 (8,6%)
java.util.HashMap$Entry[]		4.325.584 B (9,8%)	486.293 (4,6%)
java.util.HashMap$Entry		3.176.352 B (7,2%)	1.058.996 (10,1%)

Wicket 1.4.20



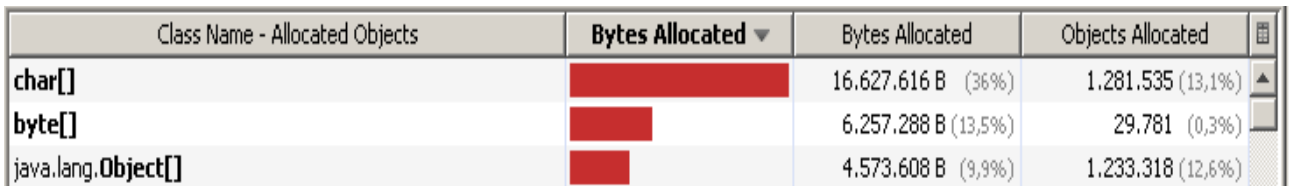
Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		15.525.184 B (38,4%)	982.107 (16,6%)
byte[]		7.849.680 B (19,4%)	30.683 (0,5%)
java.lang.Object[]		3.408.192 B (8,4%)	873.882 (14,7%)

Wicket 1.4.20 session storage



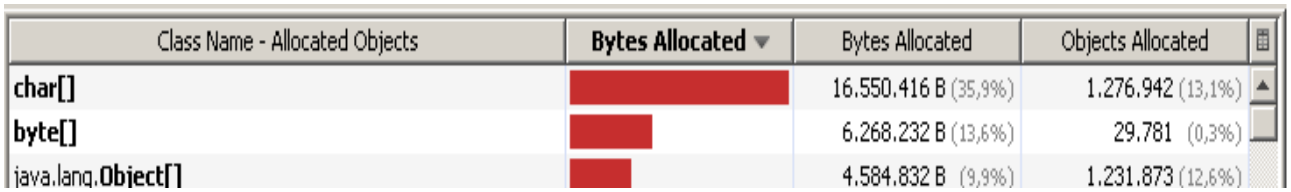
Class Name - Allocated Objects	Bytes Allocated ▼	Bytes Allocated	Objects Allocated
char[]		16.595.720 B (52,7%)	1.008.194 (20%)
java.lang.String		2.108.208 B (6,7%)	656.357 (13%)
byte[]		1.486.168 B (4,7%)	8.987 (0,2%)

Wicket 1.5.5



Class Name - Allocated Objects	Bytes Allocated ▼	Bytes Allocated	Objects Allocated
char[]		16.627.616 B (36%)	1.281.535 (13,1%)
byte[]		6.257.288 B (13,5%)	29.781 (0,3%)
java.lang.Object[]		4.573.608 B (9,9%)	1.233.318 (12,6%)

Wicket 1.5.5 session storage



Class Name - Allocated Objects	Bytes Allocated ▼	Bytes Allocated	Objects Allocated
char[]		16.550.416 B (35,9%)	1.276.942 (13,1%)
byte[]		6.268.232 B (13,6%)	29.781 (0,3%)
java.lang.Object[]		4.584.832 B (9,9%)	1.231.873 (12,6%)

Comparison table

	Total Bytes Allocated	**Objects Allocated**
MyFaces 2.1.7	14'498.532	3'062.226
Mojarra 2.1.7	44'058.840	10'485.108
Wicket 1.4.20	40'430.166	5'916.307
Wicket 1.4.20 session	31'490.929	5'040.970
Wicket 1.5.5	46'187.822	9'782.709
Wicket 1.5.5 session	46'101.437	9'747.648

In this case, MyFaces uses a lot less memory to do the same task and the memory used between Mojarra and Wicket 1.4.20 is more or less the same. Wicket disk storage requires more memory than session storage, because the pages need to be saved and restored from disk. Most memory is allocated by Wicket 1.5.5, but it is interesting that there is no significant difference between disk storage and session storage.

From a memory perspective, MyFaces is the winner but note that allocating more or less memory doesn't say much because part of this allocated memory could have a short life, but it is relevant if the web server has not a lot of memory. In a few words, the less bytes and objects allocated the better.

To see how this effect impacts performance, a simple experiment was done:

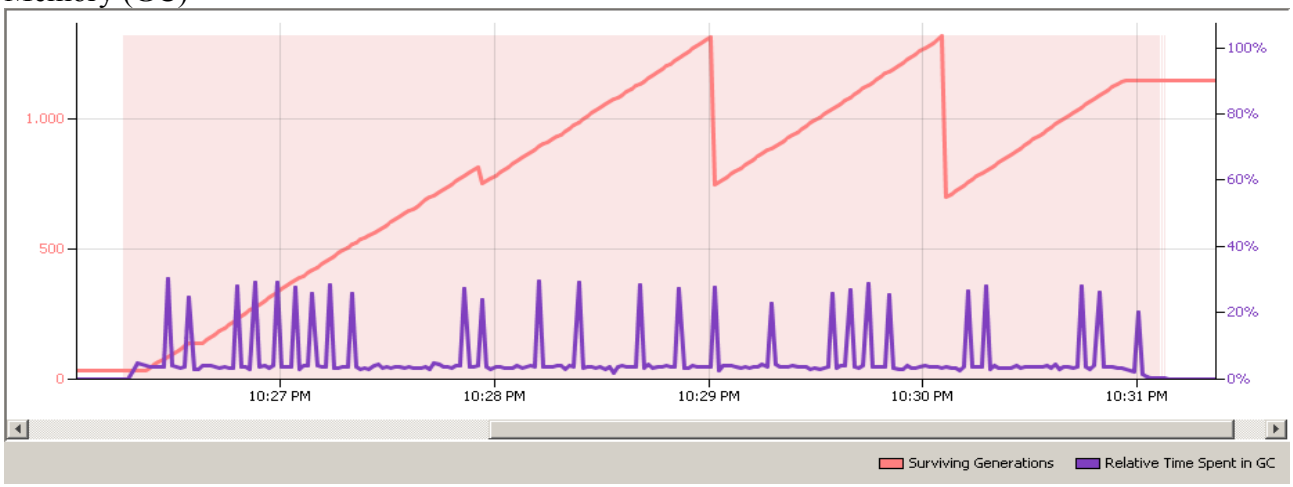
CONFIGURATION	WARMUP	EXPERIMENT
Processor: AMD Phenom x4 Speed: 2300 MHz Server: Apache Tomcat 7.0.26 JVM version : oracle jdk1.6.0_30 JVM Options : -Xms32m -Xmx32m -server	loop.count: 200 thread.count: 40 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 20 thread.deviation: 10 rampup time: 1s Http Request Client : Httpclient4	loop.count: 1000 thread.count: 40 include.logout : 1 include.delete: 1 booking.count: 1 include.ajax: 1 thread.delay: 0 thread.deviation:0 rampup time: 1s Http Request Client : Httpclient4

The idea was to just reduce the memory to the minimum possible level and increase the thread count to 40, which is more than enough to fill the memory and trigger the GC frequently. In this case only MyFaces 2.1.7, Wicket 1.4.20 using session storage and Wicket 1.5.5 were considered, because the other ones will just give some values in between. Netbeans Profiler was attached to the server and here are the results:

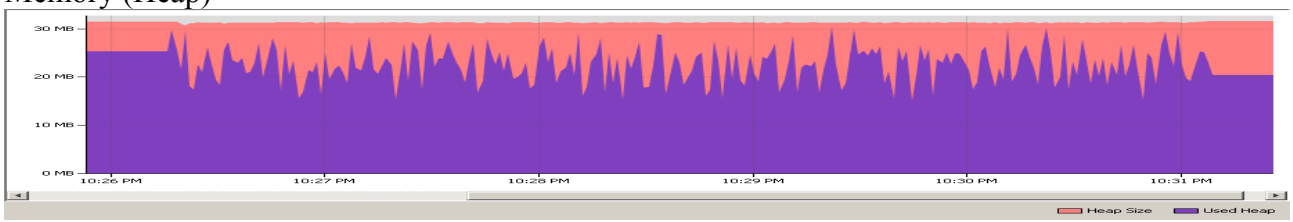
MyFaces 2.1.7

Time spent: 5 min 19 sec

Memory (GC)



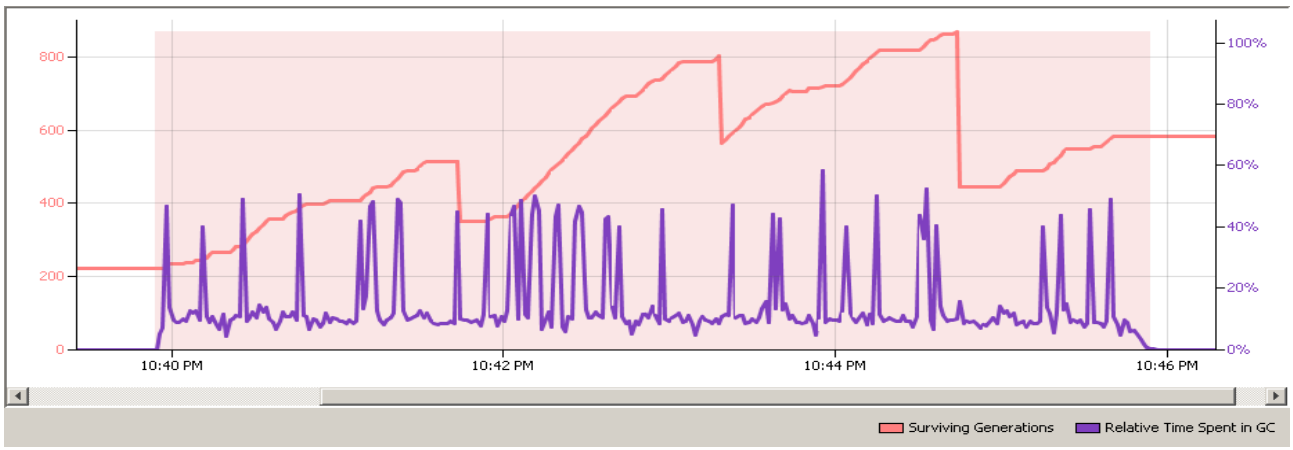
Memory (Heap)



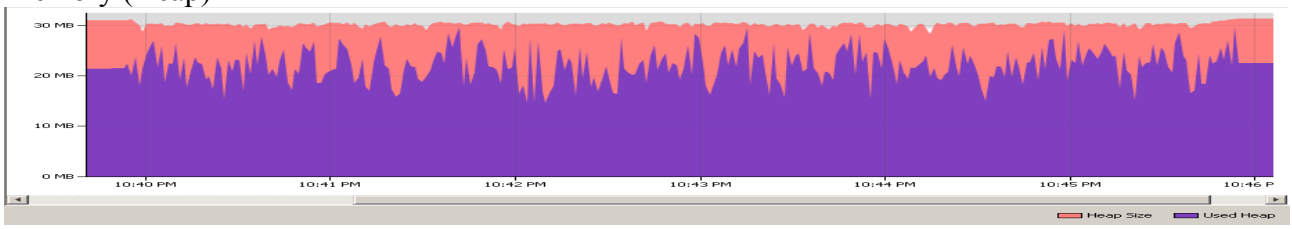
Wicket 1.4.20 using session storage

Time spent: 6 min 37 sec

Memory (GC)



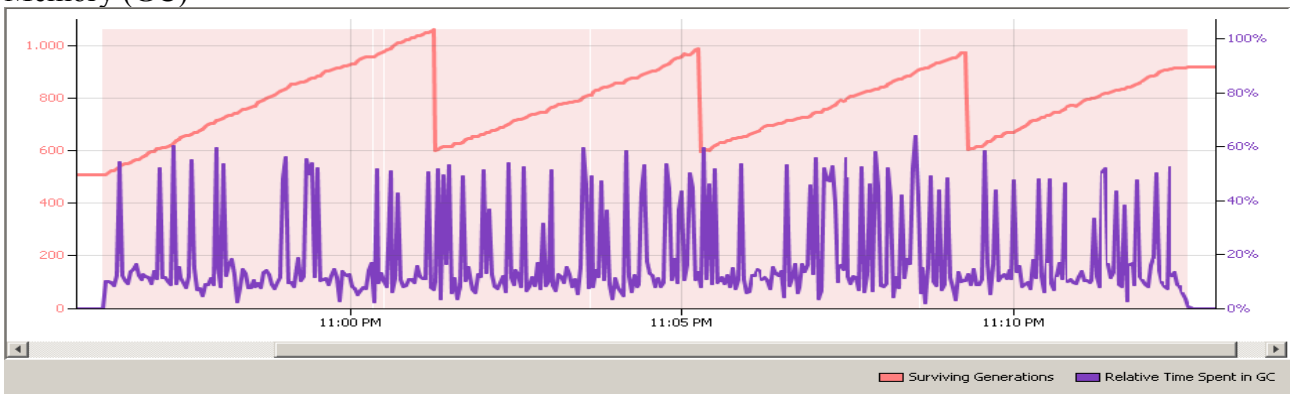
Memory (Heap)



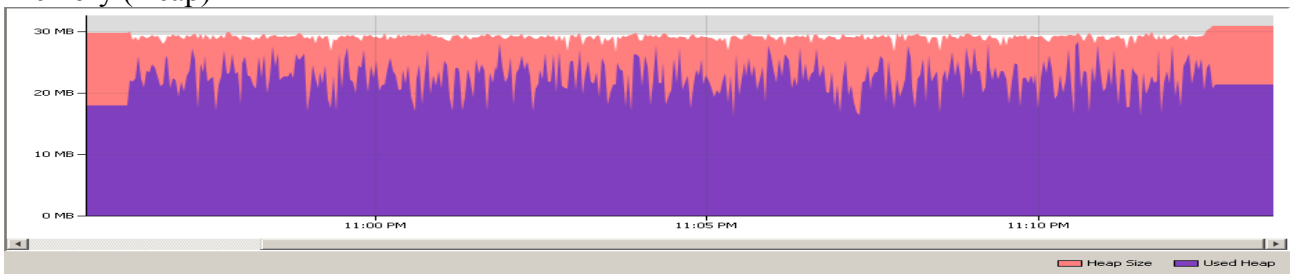
Wicket 1.5.5 (disk storage)

Time spent: 16 min 39 sec

Memory (GC)



Memory (Heap)



As expected, MyFaces does very well in this comparison because it uses less memory and allocates fewer objects. It only takes 5 min 19 sec to complete the task against Wicket 1.4.20 using session storage that takes 6 min 37 sec and 1.5.5 with 16 min 39 sec.

IV. Session Size

The last factor to take into account is how the session size is affected by the web framework. In other words, the idea is to check how much overhead a web framework imposes over session storage. This could have different implications like:

- If the session size is big, and it is stored in memory only, it is possible to exhaust the available memory, and it will limit the number of concurrent users for your system.
- If session size is big, and some persistence solution is used to store session information in some centralized place or share between servers (like in a cluster for example), more CPU is used because it will take longer to serialize/deserialize and transmit the necessary information across servers.

The final effect perceived will vary according to your particular deployment configuration, butof course keeping a small session size is better.

Since we are checking different web frameworks, some considerations:

- JSF uses a map to store views, but Wicket uses a page map and can generate stateless pages. Both frameworks use different strategies in this implementation detail, and the data gathered should be analyzed according to that. A plain comparison is not enough.
- Note for the analyzed example, wicket uses a StatelessForm for login, but a Form for creating bookings. StatelessForm means the overhead in session will be 0 in that case, but note JSF can be customized doing some changes to allow stateless pages/views too.
- It is interesting here to check how session size grows in JSF 2.1 and how Partial State Saving (PSS) algorithm introduced in JSF 2.0 has improved this situation.
- Wicket by default provides a strategy to store pages on disk, but can be setup to use the session map only. Both strategies have different implications, but in this case session mode is the one to be taken as reference.
- The session size in memory and the session size once serialized are two different measurements, in this case the session size in memory is more interesting. In JSF, options for compressing or serializing state have been disabled.
- The test application has been designed to keep the same information in session, so the differences can be attributed to the web framework implementation.

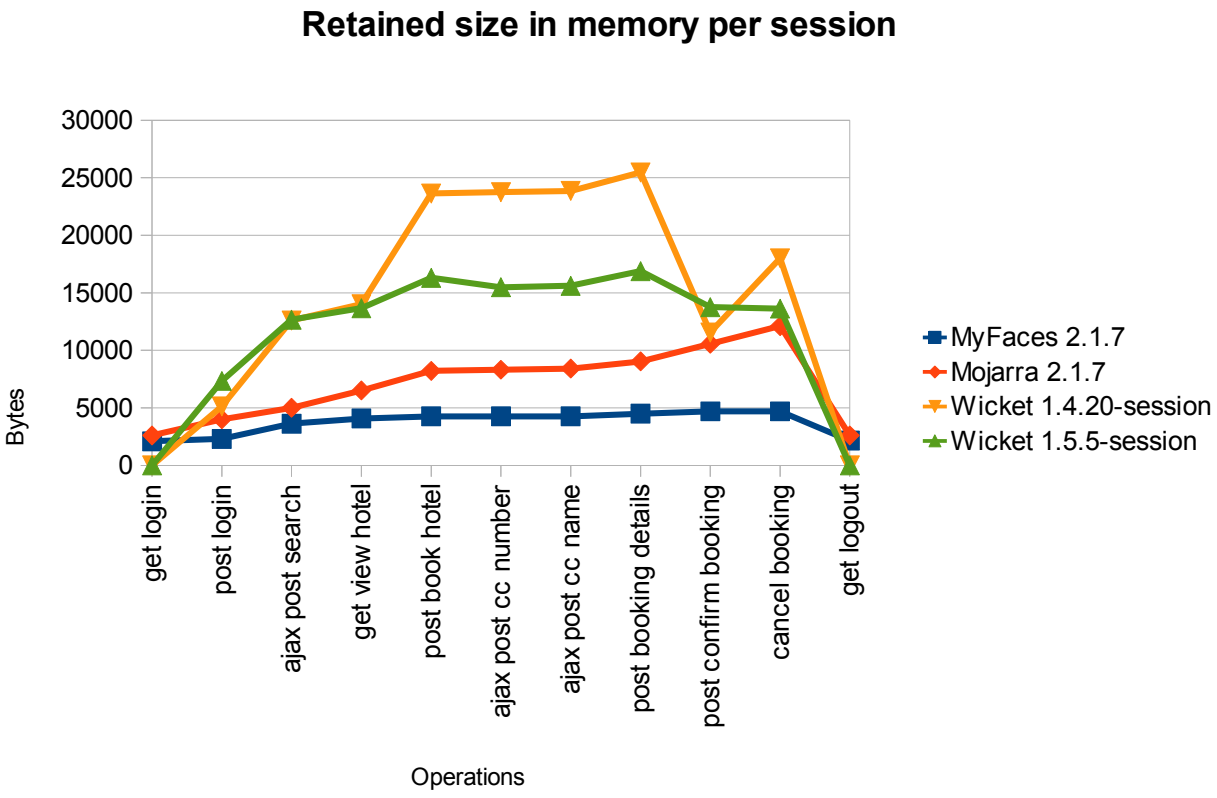
YourKit Profiler allows to see the retained size of a object, or in other words, represents the amount of memory that will be freed by the garbage collector when this object is collected. This feature is ideal to see the session size, just looking for org.apache.catalina.session.StandardSession retained size. Also, looking at the retained size of org.apache.myfaces.renderkit.ServerSideStateCacheImpl$SerializedViewCollection it is possible to see a measure of how much of the session size is used to store views.

In JSF, to ensure an accurate comparison, the number of views to store in the session has been set to 10, being used in 8 slots. Here are the results:

Retained size in memory per session (in bytes):

Session Size	MyFaces 2.1.7	Mojarra 2.1.7	Wicket 1.4.20	W 1.4.20-session	Wicket 1.5.5	W 1.5.5-session
get login	2088	2600	0	0	0	0
post login	2304	4008	2464	5128	5832	7336
ajax post search	3616	5008	3104	12600	11160	12672
get view hotel	4080	6512	3048	14008	12128	13664
post book hotel	4264	8216	3144	23632	14704	16304
ajax post cc number	4264	8312	3144	23744	13856	15480
ajax post cc name	4264	8408	3144	23856	13968	15608
post booking details	4496	9032	3144	25464	15160	16880
post confirm booking	4696	10576	3144	11552	11968	13752
cancel booking	4696	12104	3144	18008	11800	13624
get logout	2152	2600	0	0	0	0

Take a look at the difference between wicket using disk storage (by default) and wicket using session map only. There is a significant overhead involved if you compare it against JSF. Note that in the JSF case, since the number of views to store in session is 10, we can see how session grows more softly than Wicket.



Without doubt, MyFaces imposes the lowest overhead. Note according to the issue WICKET-2928, stateless ajax for Wicket is not provided by default (by a lot of well founded reasons), so there is no way to skip this problem using stateless pages.

Estimated overhead in memory per session (in bytes):

Session Size	MyFaces 2.1.7	Mojarra 2.1.7		W 1.4.20-session	Wicket 1.5.5	W 1.5.5-session
get login	424	1008		0	0	0
post login	464	2232		2664	3368	4872
ajax post search	1136	2592		9496	8056	9568
get view hotel	1560	4048		10960	9080	10616
post book hotel	1600	5608		20488	11560	13160
ajax post cc number	1600	5704		20600	10712	12336
ajax post cc name	1600	5800		20712	10824	12464
post booking details	1640	6112		22320	12016	13736
post confirm booking	1744	7680		8408	8824	10608
cancel booking	1744	9088		14864	8656	10480
get logout	512	1008		0	0	0

Note the values for wicket are estimated, based on the usage of session in wicket 1.4.20 by default and subtracting it from the retained size. You can see how the difference between JSF and Wicket is wide in this part. Each time a page is added to the storage, the session size grows, but not in the same magnitude as JSF.

JSF average view size is very small compared to the user session size, imposing a reasonable overhead. MyFaces algorithm does the best job with the state, but to appreciate this point better it is necessary to study in depth the factors that make the state bigger when the PSS algorithm is enabled.

PSS algorithm is based on the idea of differentiating between the "initial state" of a view and the "delta". If it is possible to restore the "initial state" of a view, it is just necessary to save the "delta" or the changes between the initial state and the current view to restore it correctly. But if a part of the component tree is changed drastically, the whole state of that part needs to be saved as delta, because it is not possible to calculate any initial state from that part.

There are some common use cases in JSF that are affected by this:

- Use of <composite:insertChildren> and <composite:insertFacet> tags: the use of these tags cause a relocation of all related components into a new location. In MyFaces 2.0.2 and later, a different algorithm was proposed to overcome this problem and others.
- Use of <ui:include src="#{...}">, <ui:define template="#{...}">, <c:if ...> and <c:choose> : in this case a part of the component tree is updated dynamically, based of the value returned by an EL expression. A new feature was added in MyFaces core 2.1.6/2.0.12 to overcome this problem, storing the result of the expression and including it in the initial state. See MYFACES-3451 for details.

To understand and see the effect of these features, an example webapp that shows the state size in some selected use cases was provided. Here are the results:

Without PSS (Partial State Saving)

This is the maximum state size obtained to save the whole tree.

	MyFaces 2.1.7 Without PSS			Mojarra 2.1.7 Without PSS		
Example	Initial Size [bytes]	Validation Error [bytes]	Succesful Postback [bytes]	Initial Size [bytes]	Validation Error [bytes]	Succesful Postback [bytes]
HelloWorld	6611	6666	6697	9051	9110	9151
Composite Component HelloWorld	8419	8504	8535	10877	10931	10972
Insert Children HelloWorld	7967	8022	8053	10285	10411	10452
Insert Children HelloWorld With More Components	9015	9070	9101	13308	13866	13907
Dynamic Include B HelloWorld	8040	8092	8123	11387	11415	11456
Dynamic Include C HelloWorld	9152	9214	9235	14827	14855	14896

Mojarra 2.1.7 has bigger values in this comparison, but note this is just a reference, to see how big a view can be without PSS.

With PSS (Partial State Saving)

	MyFaces 2.1.7 With PSS			Mojarra 2.1.7 With PSS		
Example	Initial Size [bytes]	Validation Error [bytes]	Succesful Postback [bytes]	Initial Size [bytes]	Validation Error [bytes]	Succesful Postback [bytes]
HelloWorld	46	349	449	1836	2014	2138
Composite Component HelloWorld	46	356	456	2074	2601	2725
Insert Children HelloWorld	46	356	456	2098	3154	3278
Insert Children HelloWorld With More Components	46	356	456	2100	3805	3929
Dynamic Include B HelloWorld	788	1013	1113	6422	6447	6504
Dynamic Include C HelloWorld	788	1013	1113	9130	9155	9212

Note in MyFaces the state does not grow with more components ("Dynamic Include C" and "Insert Children HelloWorld With More Components"). The impact of composite components on the state is minimal. In conclusion, MyFaces definitively does a very good job in this regard, keeping the state size small. Even when it is a simple view (HelloWorld) without any of the problems mentioned before, the difference is noticeable.

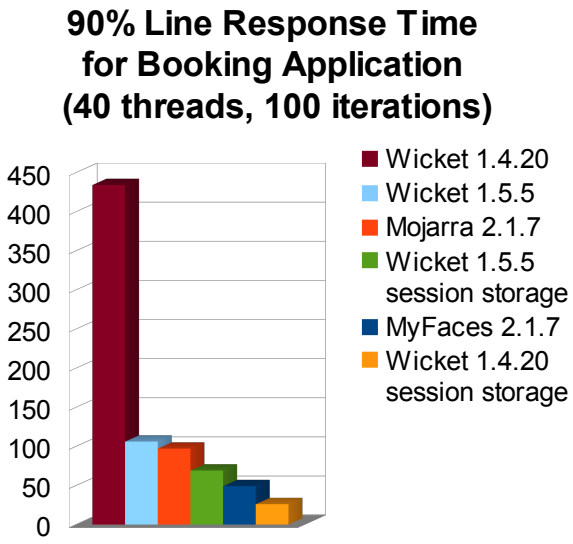
V. Conclusion

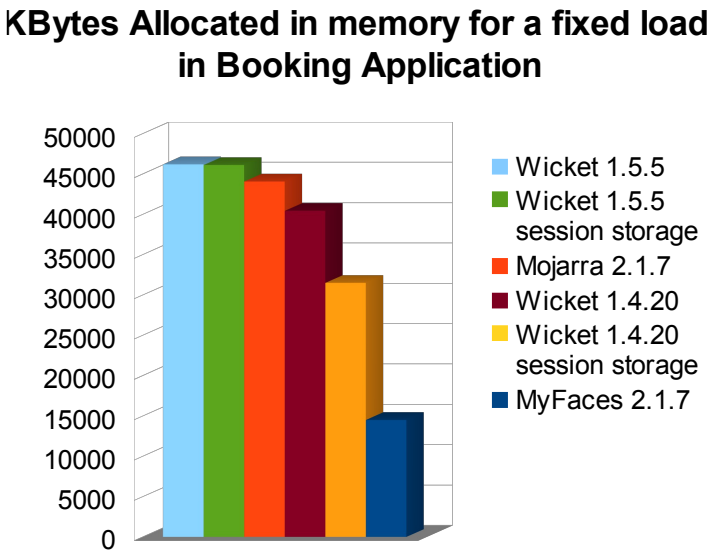
Definitively JSF 2.0 was a big move in the right direction. JSF session size overhead is significantly lower than Wicket. MyFaces Core 2.1.7 looks very well from a performance perspective against a web framework like Wicket. In the end, MyFaces provides the better trade-off between CPU usage and session size. The bet of Wicket on disk storage is good, but the JSF bet on partial state is even better. MyFaces also provides the lowest memory usage, which allows it to perform better in environments with limited CPU and memory.

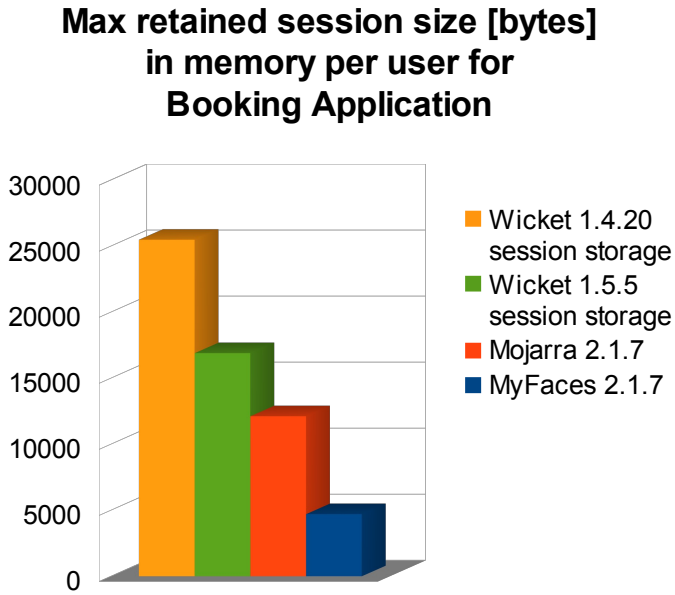
Comparing Wicket 1.4 vs Wicket 1.5, it seems that the improvement in its programming model is taking a toll. In my opinion it is something completely reasonable. A good web framework needs to balance different aspects in order to keep evolving over the time. Including more assisting code

usually means lower response times, but it could improve developer productivity.

Taking into consideration all previous observations, here are the final graphs.







Please note these graphs cannot be generalized, and they depend strongly on the test application, but they are the best indication available at the moment. The test application uses ajax to update the view, a typical login/logout case, uses a datatable and has a wizard use case.

This is not the end of the story. Web frameworks will keep improving and the hope is this information can be useful to find new ways to enhance them (community over code is the Apache way). Performance is just one aspect that you have to consider when choosing a web framework; usually it is necessary to strike a balance between this and several other aspects.

VI. Acknowledgement

Thanks to all people who help to review and provided ideas and feedback for this work in MyFaces community: Martin Kočičák, Gerhard Petracek, Mark Struberg and Martin Marinschek.

VII. References

The initial comparison between Seam JSF vs Wicket code can be found on:

1. http://ptrthomas.wordpress.com/2009/01/14/seam-jsf-vs-wicket-performance-comparison/
http://perfbench.googlecode.com/svn/trunk/perfbench/

The updated code, configuration used, detailed documentation and experimental data can be found in the internet (github) at:

2. https://github.com/lu4242/performance-comparison-java-web-frameworks

Apache Projects used in this comparison:

3. Apache MyFaces http://myfaces.apache.org/
4. Apache Wicket http://wicket.apache.org/
5. Apache Tomcat http://tomcat.apache.org/
6. Apache JMeter http://jmeter.apache.org/

In this comparison YourKit and Netbeans profiler was used:

7. YourKit http://www.yourkit.com/
8. Netbeans Profiler http://netbeans.org/