Contents

1 Introduction 3

2 Test Setup 4
- 2.1 Server Details . . . 4
  - 2.1.1 Database . . . 4
- 2.2 Client Details . . . 5
- 2.3 Details of test plan . . . 5

3 Application Details 6
- 3.1 Search . . . 6
- 3.2 Search results . . . 8
- 3.3 Checkout . . . 10
- 3.4 Payment . . . 10
- 3.5 Load booking . . . 12
- 3.6 Get booking . . . 12

4 Test results 14
- 4.1 Response times . . . 14
- 4.2 Throughput . . . 15
- 4.3 Number of requests in the web application . . . 16
- 4.4 Utilization of the Tomcat-container and system . . . 17
- 4.5 Throughput of database . . . 18

5 Discussion 20
- 5.1 Application performance . . . 20
  - 5.1.1 Analysis of the system utilisation . . . 20
  - 5.1.2 Analysis of RTT & the application capacity . . . 21
  - 5.1.3 Analysis of the different steps in the operation . . . 22
  - 5.1.4 Error analysis . . . 22
- 5.2 Possible performance improvements . . . 23
  - 5.2.1 Connection pooling . . . 23
  - 5.2.2 Database interaction . . . 23

6 Conclusion 24

1 Introduction

This report provides test results and analysis of the performance of the Hotel Booking application developed in assignment 2 in the course COMP9321. We start out by describing how the test was set up in order to produce consistent and measurable results. Then we describe how the relevant application components are designed and how they work in order to serve user requests. The next part shows the values of the test metrics requested in the assignment specification. Lastly, we analyse the performance of the application based on our test results, and we discuss possible performance improvements.

2 Test Setup

A mid 2011 MacBook Air running Apache Tomcat 7.0.42 was used as the server. The client, running JMeter, was an early 2011 MacBook Pro.

2.1 Server Details

Hardware variable	**Value**
CPU	1.7 GHz Intel Core i5
RAM	4 GB 1333 MHz DDR3
SSD	128 GB

Table 1: Hardware details.

Operating variable	**Value**
Operating system	Mac OS X 10.9.2 (Mavericks)
Java version	1.6.0_65
JVM Server	Java HotSpot(TM) 64-Bit Server VM
JVM Server version	20.65-b04-462
JVM maximum memory	2.54 GB
Apache Tomcat Server	7.0.42
Database	MySQL
MySQL version	5.5.34

Table 2: Operating enviroment.

2.1.1 Database

The application uses MySQL 5.5.34 as database server and Hibernate 4.3.5 as the object relational mapping engine. Hibernate ships with default settings for handling database connections and connection pooling. We have kept these settings as they did not cause unacceptable performance.

Connection pool variable	**Value**
Minimum	1
Maximum	20

Table 3: Connection pooling.

2.2 Client Details

Hardware variable	Value
CPU	2.7 GHz Intel Core i7
RAM	16 GB 1333 MHz DDR3
SSD	Samsung SSD 840 EVO 250 GB

Table 4: Hardware details.

2.3 Details of test plan

The test was constructed using a MacBook Air acting as the server running Tomcat 7.0.42 and using a MacBook Pro running JMeter as the client(s). The tests were run over a computer-to-computer network to minimize disturbance from uncontrolleable factors. The following scenario was to be simulated:

1. The user searches for two rooms with check in date 1st of June 2014 and check out date 8th of June 2014. The user selects a queen and a twin room, and then proceeds to confirm the booking.

2. The user then loads the unique page for his/her booking using the unique PIN and system-generated URL provided in the email sent by the application.

A delay of five seconds was applied to each user action.

The test was conducted for a wide range of concurrent users: 10, 50, 100, 250, 500, 750, 1000, 1250, 1500, 1750 and 2000. The test was constructed using JMeter testing software. By using its 'Thread Group' feature we constructed a set order of HTTP requests to be sent to the server in order to replicate actions taken by actual users. The test was conducted such that the user checks his/her booking directly after it's been made. Thus, new bookings are still being made when the first users access their respective bookings. We believe this yields a better and more realistic test than if the the two test phases were separated.

The requested hotel has a capacity of 100 queen rooms and 100 twin rooms. Thus, the first 100 threads started will result in successful bookings. Subsequent threads will not successfully book rooms as the room capacities have been reached. The next users will thus fail somewhere in the process. Most of the threads will fail after the initial search because no Queen rooms or Twin rooms are available. Some users will also fail before submitting payments because the system checks available capacity before payment is processed. In this way no payments happen for bookings that cannot be met by the hotel.

Originally we wanted to use the MacBook Pro as the server machine given its superior technical specs, but when we conducted the first tests we discovered that JMeter could not run on the MacBook Air for high user numbers. We therefore switched the two

machines, so that the MacBook Pro represented clients while the MacBook Air worked as the server. We were able to raise the number of users to 2000 before JMeter started freezing again. We tried to mitigate the problems by tuning available memory for both JVM and JMeter, but we weren't successful. Anyways we believe we have a good amount of data to describe the performance of our web application.

Exact testing procedure:

1. Empty database of existing bookings
2. Reload application with PSI Probe application
3. Load application in web browser
4. Restart JMeter
5. Run JMeter tests

3 Application Details

The booking process in out application is broken down into five separated stages.

1. Search
2. Search results
3. Check out
4. Payment
5. Confirmation

An activity diagram for the booking process is shown in figure 5. This diagram is a little simplified. Most importantly, we have not included the "controller" in the digram. In the actual application each request is first sent to the controller, which selects the appropriate filter chain to pass the request through, and the command to perform if the request successfully passes all the validation filters.

3.1 Search

The user selects check in date, check out date, city and a required number of rooms. The user may also choose a maximum price per room per night or leave it blank. When the user hits the search button calls are made to the database in order to display the correct search results.

The search process is shown in Figure 1. First a call is made to the database to find all room types. Second, a call is made to the database to find all applicable discounts for the selected hotel between check-in and check-out. Third, a call is made to find out if

any of the dates within the booking period are peak period dates. Based on the results of these three calls, the application computes the price for each of the dates between check-in and check-out, for each of the different room types.

Further, the application computes the free capacity of the selected hotel in the booking period. First, a call is made to the database to find the total capacity of each room type in the selected hotel. Second, a call is made to the database to find all room bookings of the selected hotel. Based on these two calls, the free capacities between check-in and check-out, for each of the different room types in the selected hotel, are computed.

A specific room type is then added to the search results if it satisfies the following two criterions:

$$FreeCapacity_{roomType} > 0 \tag{1}$$

$$\frac{PricePerNightSoFar + PricePerNight_{roomType}}{NumOfRoomsSelectedSoFar + 1} \leq MaxPrice \tag{2}$$

The table (5) below shows the number of values that may be returned for each of the five calls to the database in the search process. Obviously, more work needs to be done in the database when its is densely populated.

Database calls	**(Min, Max) no. of returned elements**
getAllRoomTypes()	(5, 5)
getDiscounts()	(0, no. of discounts satisfying parameters)
getPremimums()	(0, no. of days in booking)
getCapacities()	(5, 5)
getRoomBookings()	(0, no. of bookings for the selected hotel)

Table 5: Table of database calls and their return values.

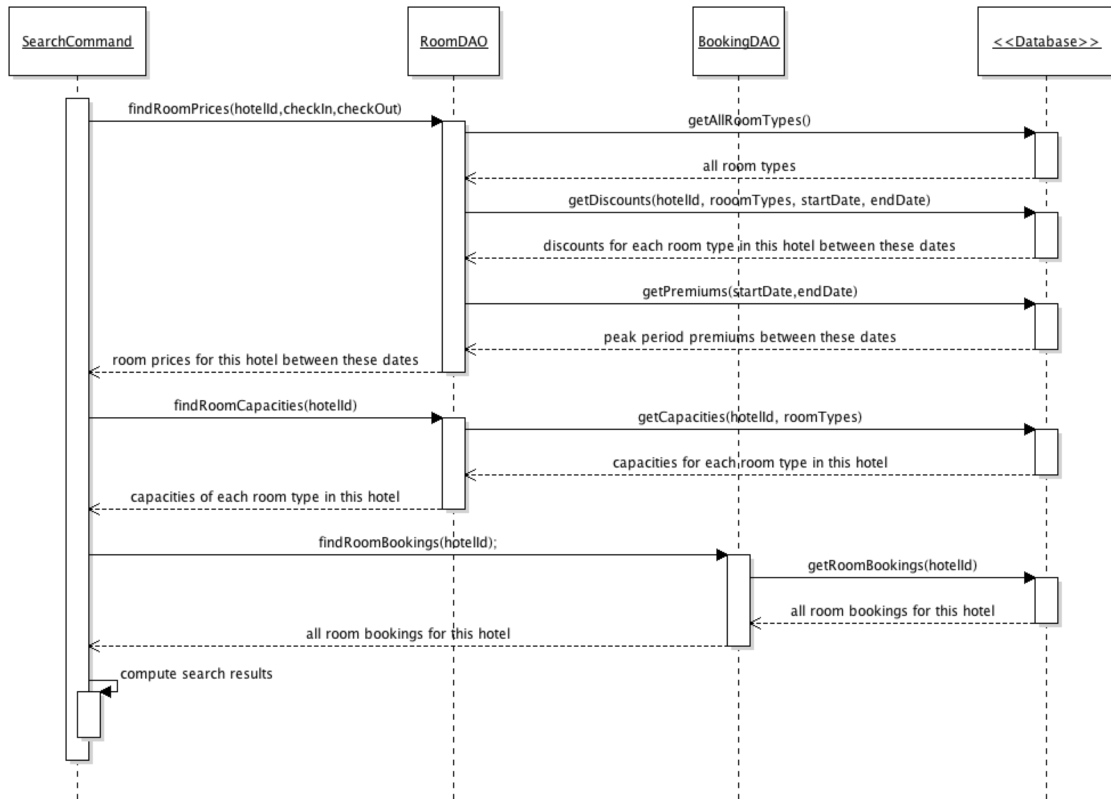


Figure 1: Sequence diagram for the search process

3.2 Search results

The search results are distributed across i pages, where i equals the number of rooms required by the user. At each page, the user selects a room, and wether or not she wants an extra bed in this room (see figure 2a and 2b). When the user has selected a room, the room capacities are updated. Next, if there are more rooms to be added, the user is taken to the next "select room" page. Prior to being redirected to this page, new results are computed based on the criterions in equation 1 and 2. This computation can most easily be described using an example:

Example 3.2.1

Given the following search parameters:

- Max Price = 200
- Number of rooms = 2
- Number of nights = 1

Assumes that there are no discounts, no peak period, and sufficient capacity.

First search result computation:

- $Price_{PerNightSoFar} = 0$
- $Price_{single} = \frac{0+80}{0+1} = 80 \leq 200 \implies$ Add to search results
- $Price_{twin} = \frac{0+120}{0+1} = 120 \leq 200 \implies$ Add to search results
- $Price_{queen} = \frac{0+120}{0+1} = 120 \leq 200 \implies$ Add to search results
- $Price_{executive} = \frac{0+180}{0+1} = 180 \leq 200 \implies$ Add to search results
- $Price_{suite} = \frac{0+300}{0+1} = 300 \leq 200 \implies$ Do not add to search results

The user selects "single room" at the first select room page.

Second search result computation:

- $Price_{PerNightSoFar} = 80$
- $Price_{single} = \frac{80+80}{1+1} = 80 \leq 200 \implies$ Add to search results
- $Price_{twin} = \frac{80+120}{1+1} = 100 \leq 200 \implies$ Add to search results
- $Price_{queen} = \frac{80+120}{1+1} = 100 \leq 200 \implies$ Add to search results
- $Price_{executive} = \frac{80+180}{1+1} = 130 \leq 200 \implies$ Add to search results
- $Price_{suite} = \frac{80+300}{1+1} = 190 \leq 200 \implies$ Add to search results

---

During selection of rooms, the application makes no calls to the database. The second search result computation uses the values found during the initial search.

Search / Search results / Check out / Payment / Confirmation

Room 1

Room type	Description	Free capacity	Price per night	Extra bed	Select
Single	bla	15	$ 70.0	NO	Select
Twin	bla	10	$ 100.0	NO	Select
Queen	bla	10	$ 120.0	NO	Select
Executive	bla	5	$ 150.0	NO	Select
Suite	bla	2	$ 300.0	NO	Select

0/3

Back to search

(a) Selecting the first room

Room 2

Room type	Description	Free capacity	Price per night	Extra bed	Select
Single	bla	14	$ 70.0	NO	Select
Twin	bla	10	$ 100.0	NO	Select
Queen	bla	10	$ 120.0	NO	Select
Executive	bla	5	$ 150.0	NO	Select
Suite	bla	2	$ 300.0	NO	Select

1/3

Back to search

(b) Selecting the second room

3.3 Checkout

The checkout page summarizes the user's booking request. It shows the types of rooms selected, chosen city, check in date, check out data and pricing information. Again, as with the select room pages, no database interaction is taking place at this stage of the booking process. The user can either confirm and go to payment or cancel the booking.

3.4 Payment

The payment page presents a form for filling in your email and payment details. When the "pay now" button is clicked, the free capacity of the hotel is confirmed by making another call to the database. This is done to ensure that the hotel still has enough free capacity for this booking (the capacity may have been reduced since the user made the search). Further, the booking is written to the database and the user is taken to the confirmation page. Additionally, a separate thread for sending the confirmation email is

started. The payment process is shown in figure 3.

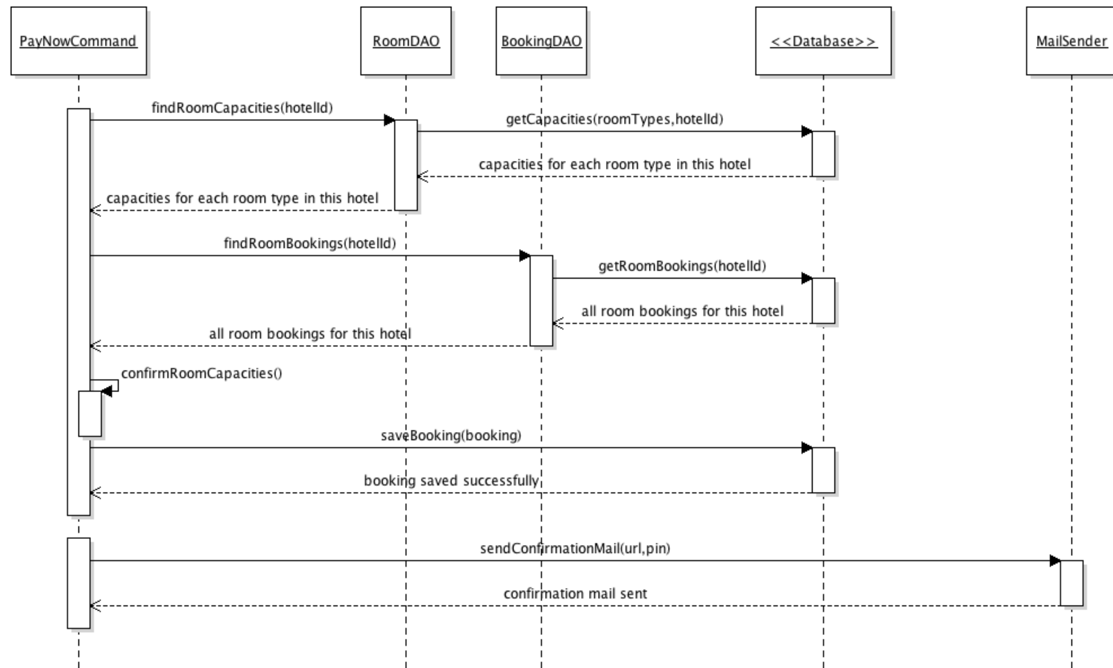


Figure 3: Sequence diagram for the payment process

3.5 Load booking

When the user clicks the link provided in the email sent after a successful booking, the user is taken to the load booking page. It consists of a single form entry prompting the user to enter his/her pin. The email link contains two parameters used in the processing of the load booking request: *url* and *action*. *url* specifies the unique booking the user wants to access, while the action parameter specifies that the load booking page should be shown to the user. The booking is then loaded from the database and into the application, so that its associated pin number and url are accessible in the application layer. This loading is also done in order to make it easier to enforce the 48 hours limitaiton on accessing the booking.

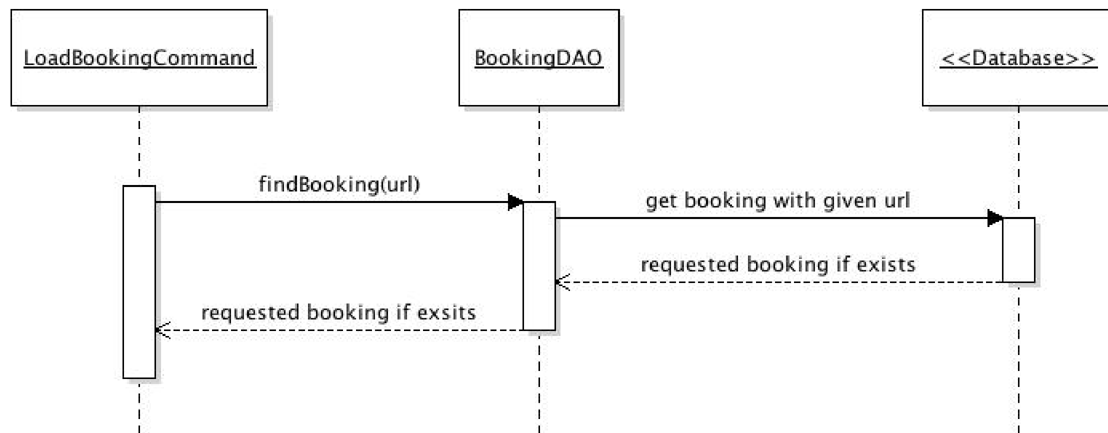


Figure 4: Sequence diagram for loading bookings

3.6 Get booking

When the user inputs a pin number in the load booking form, the loaded booking is accessed. If the pin number matches the pin number stored for the booking, the user is directed to the booking page.

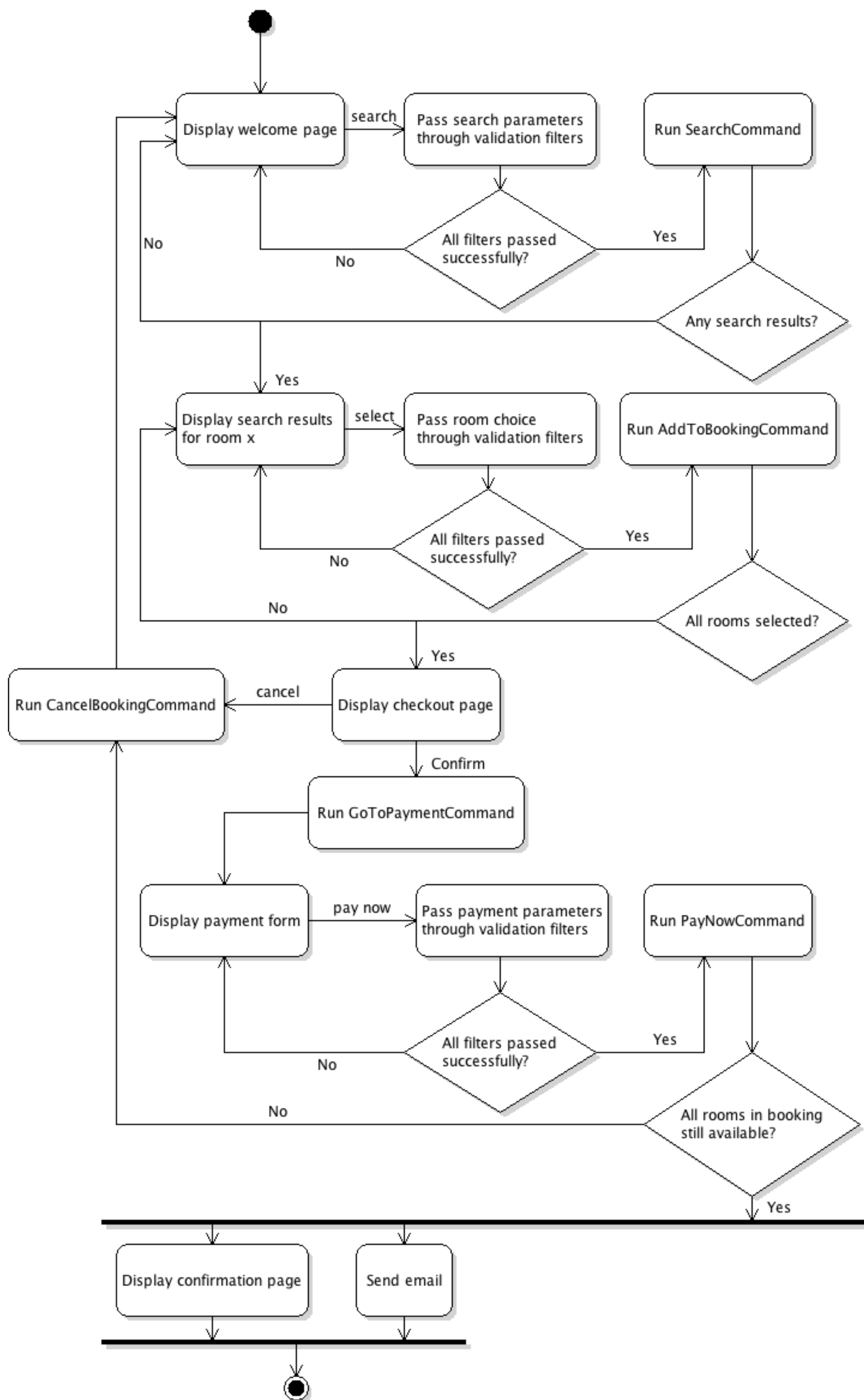
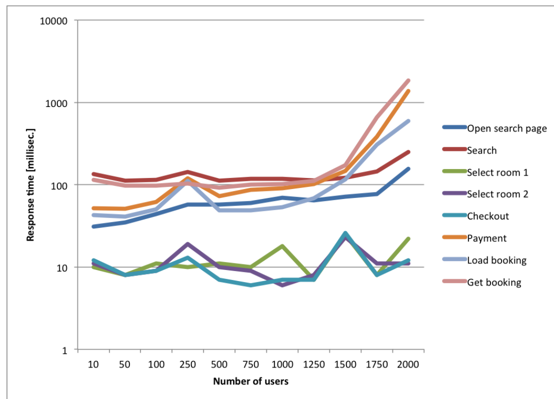


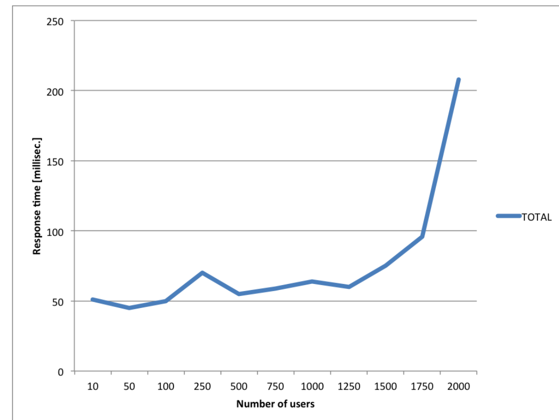
Figure 5: Activity diagram for the booking process

4 Test results

4.1 Response times

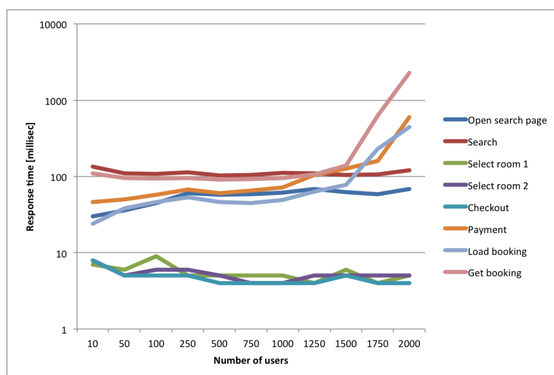


(a) Client-side mean RTT

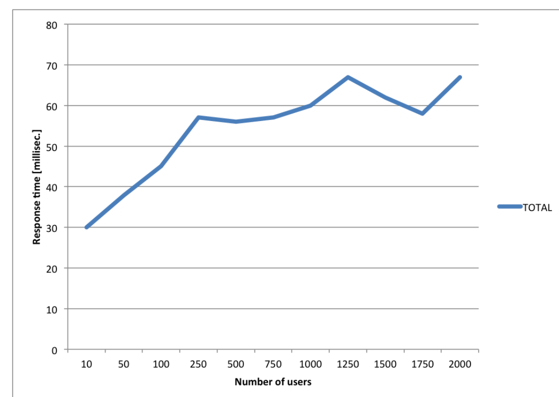


(b) Client-side mean RTT

Figure 6: Figure a shows the mean response time of each of the different steps of the booking/access booking operation, for each set of users. Figure b shows the mean response time of the complete operation, for each set of users.



(a) Client-side median RTT



(b) Client-side median RTT

Figure 7: Figure a shows the median response time of each of the different steps of the booking/access booking operation, for each set of users. Figure b shows the median response time of the complete operation, for each set of users.

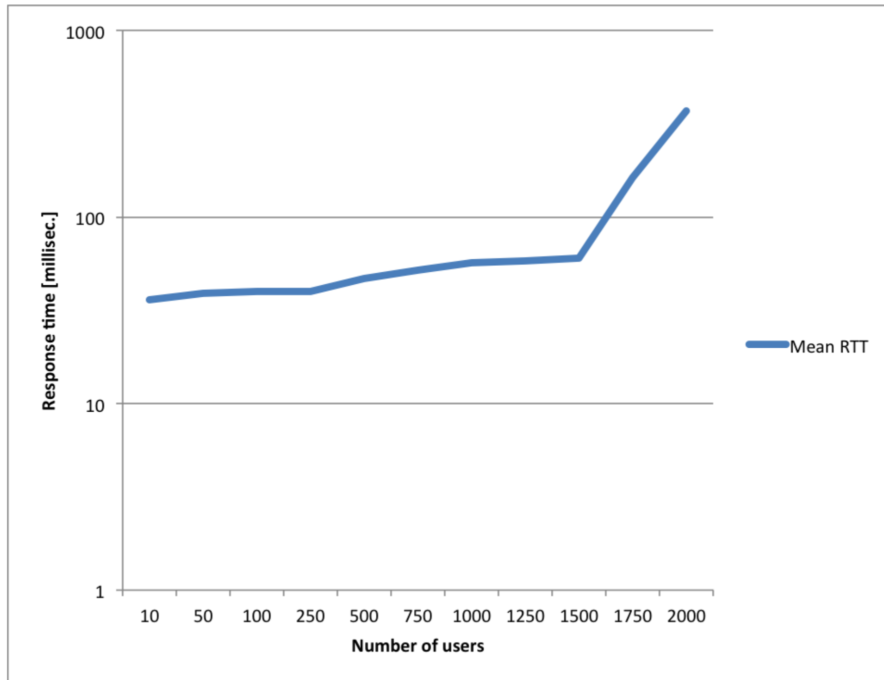


Figure 8: Server-side mean RTT

4.2 Throughput

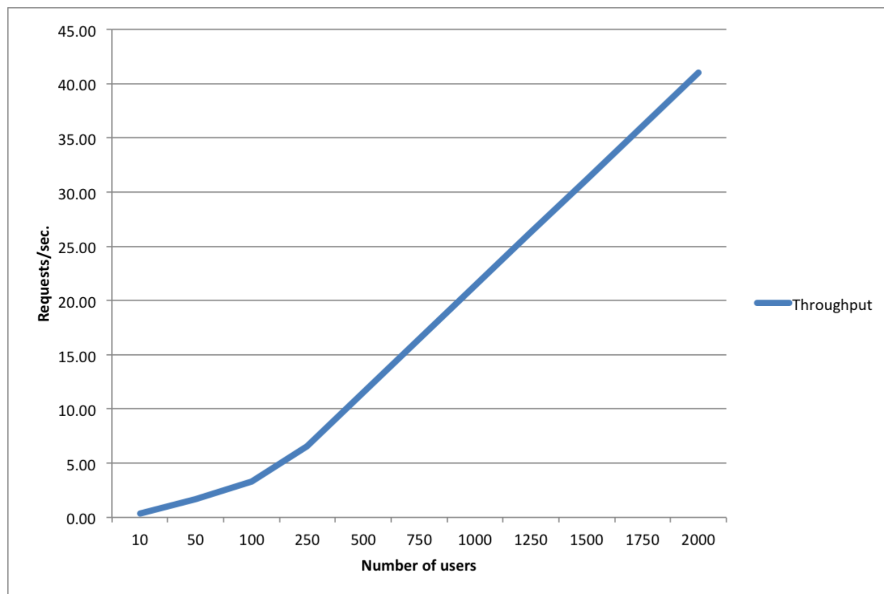


Figure 9: The client-side throughput, for each set of users.

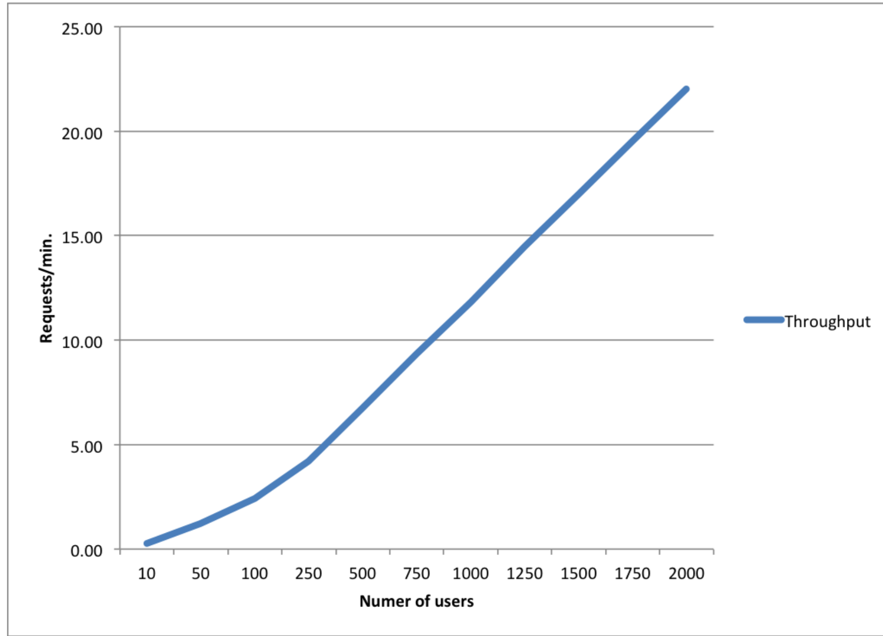


Figure 10: The server-side throughput for each set of users.

4.3 Number of requests in the web application

No. of users	R [sec]	X [req/sec]	$Q = R \times X$ [req]
10	36×10^{-3}	0.26	0.01
50	39×10^{-3}	1.21	0.05
100	40×10^{-3}	2.41	0.10
250	40×10^{-3}	4.19	0.17
500	47×10^{-3}	6.77	0.32
750	52×10^{-3}	9.34	0.49
1000	57×10^{-3}	11.83	0.67
1250	58×10^{-3}	14.47	0.84
1500	60×10^{-3}	17.03	1.02
1750	163×10^{-3}	19.54	3.19
2000	369×10^{-3}	22.01	8.12

Table 6: Number of requests in the web application (average queue length), computed using Little's law

4.4 Utilization of the Tomcat-container and system

No. of users	B [sec]	τ [sec]	$U = B/\tau$
10	3.49	307.69	1.14 %
50	15.86	330.58	4.80 %
100	32.77	331.95	9.87 %
250	61.23	309.05	19.81 %
500	98.02	310.04	31.61 %
750	141.97	310.16	45.77 %
1000	192.86	310.50	62.11 %
1250	245.09	310.70	78.88 %
1500	293.26	311.03	94.29 %
1750	540.62	311.30	173.55 %
2000	1330.41	311.68	426.85 %

Table 7: Utilization of the Tomcat container and system. B is the total processing time, given by PSI probe. τ is the time it took to run the tests (observation period).

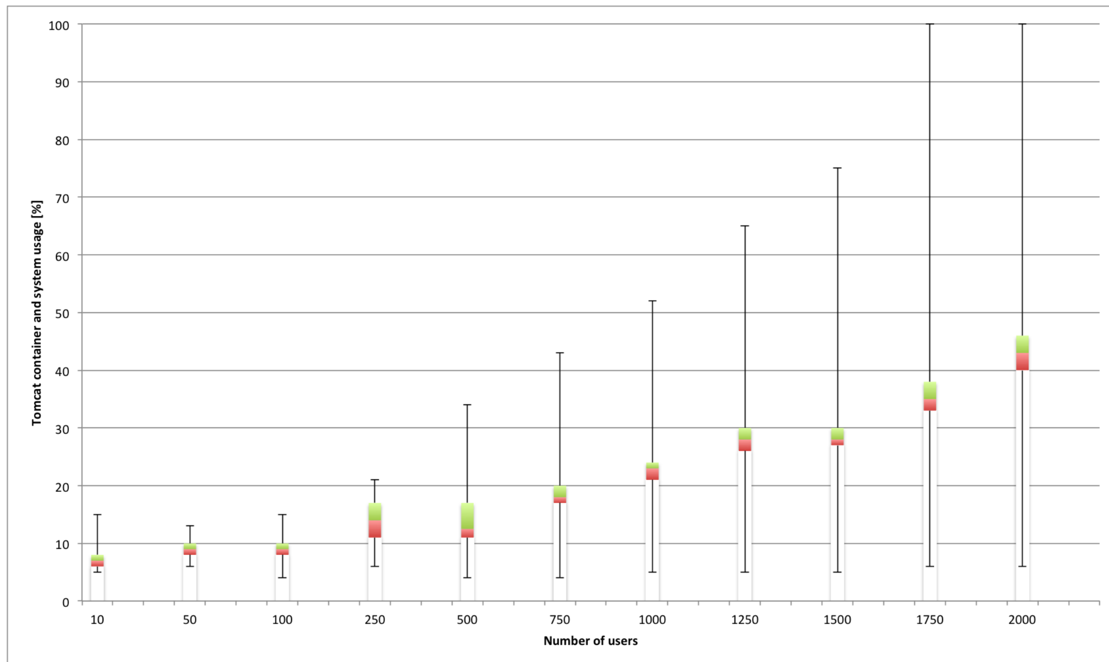


Figure 11: CPU usage per number of users

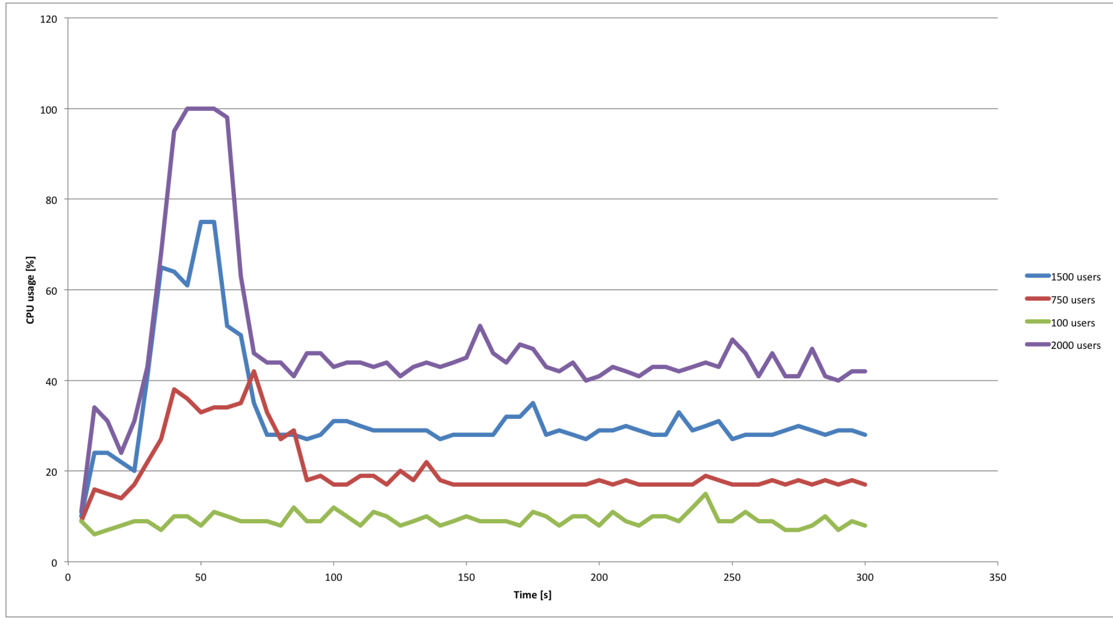


Figure 12: CPU usage per time

4.5 Throughput of database

No. of users	V [avg. no. of visits]	X_{sys} [req/sec]	$X_{db} = V \times X_{sys}$ [req/sec]
10	1.38	0.26	0.36
50	1.38	1.21	1.67
100	1.38	2.41	3.31
250	1.57	4.19	6.57
500	1.70	6.77	11.50
750	1.76	9.34	16.44
1000	1.79	11.83	21.35
1250	1.82	14.47	26.28
1500	1.83	17.03	31.18
1750	1.84	19.54	36.06
2000	1.85	22.01	41.01

Table 8: Throughput of the database, computed using the Forced Flow Law.

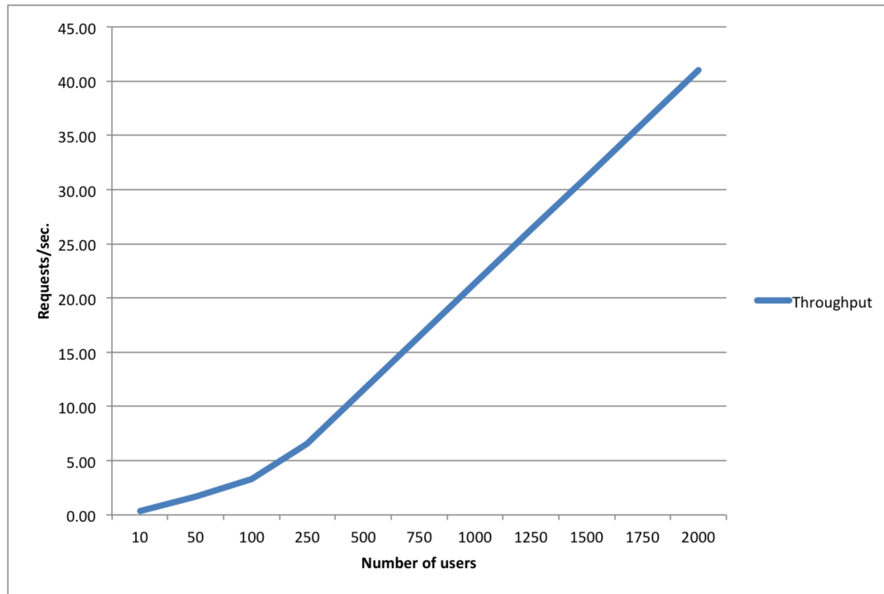


Figure 13: Throughput of the database.

5 Discussion

5.1 Application performance

5.1.1 Analysis of the system utilisation

When conducting the tests, all unnecessary processes on the server were ended. The processes running while conducting the test were the standard system processes along with the necessary server processes, such as Tomcat Server and MySQL Server. The load on the server before the tests commenced were very low, around 2% CPU usage. We therefore state that the load on the system during tests is attributable to the server responding to requests from the JMeter application running on the client.

We sampled system usage every 5 seconds during the tests. The main metric used were CPU usage. We also looked at the memory usage during the tests, but we did not have a convenient way of sampling memory usage as we did with CPU utilisation. None of the tests showed sign of memory stress or were anywhere near reaching the limit of 2GB we had assigned to the server. Memory usage were roughly around 500MB for all test cases, showing some increase under heavy load, as one would expect. As memory usage were no problem during the tests and within a reasonable limit, we conclude that our application is not causing the running system problems in terms of memory handling.

Figure 11 shows the system utilisation for different amounts of concurrent users. We see that system usage increases when additional users are added to the tests. This is no surprise. The bottom whisker shows the system usage at the very start of the test, when no or very few user threads have been initiated by JMeter. This is, as expected, the same for all tests no matter how many user will eventually be added. The green and red box is the 25th to 75th percentile (including the median). It is a measure of the mean system usage during the tests. For each test case, system usage is rather constant on average. This is evident as most of the samples is clustered in the Q1-Q3 percentile. We believe the main reason for this is the test setup. In the test setup, the first 100 users will be able to book one Queen room and one Twin room each. The rest of the users will be rejected at some point in the search process, most of them after the initial search as the requested hotel rooms are no longer available. As more and more users gets rejected compared to the amount of users able to book, the mean and median system utilization decrease caused by the lesser load on the system. Successful bookings are more expensive in terms of CPU usage than rejections. For low numbers of users (< 100), the server has no problems handling the requests and system usage is roughly the same for user amounts less than a hundred.

The top whisker shows the peak load on the system. This happens when lots of requests hit the server simultaneously from concurrent users. This is especially critical at the beginning of tests with a high number of users (> 1000) when multiple successful bookings are processed simultaneously. Successful bookings put more stress on the database than rejected requests, thus the processor is pushing more load in the first phase of the tests.

Figure 12 tells the same story. It shows how the load on the system peaks quickly

after the start of the test when lots of successful bookings are made. It also shows the consistent performance of the application when the hotel is fully booked and all booking requests are being rejected. Obviously, more user requests being rejected simultaneously puts more load on the system, as can be seen in figure 12.

5.1.2 Analysis of RTT & the application capacity

In this part we address performance metrics essential for user experience and user handling. How many users can the application handle simultaneously? What performance can users expect under small and heavy load?

RTT (round trip time) is an important metric for measuring performance from the user's viewpoint. RTT for individual requests measures the amount of time a user spends waiting between pages. The sum of these RTTs yields the time a user spends inactive during the booking process. Figure 6a and figure 6b show that RTT performance is good up to 1500 users. For user amounts larger than 1500, some steps of the operation take considerably more time. It seems that RTT grows exponentially from 1500 users and onwards, but we were not able to control this as JMeter constantly failed for more than 2000 users, even though we adjusted available memory for the JMeter and JVM. It is though beyond doubt that performance decreases in terms of RTT when the number of users exceeds 1500. Even though the median and average response time (60 ms, 75 ms) are good numbers, we recorded some outliers exceeding 5 seconds which is quite a long response time. For user numbers exceeding 1500 this patterns strengthens. Median and mean response time increase, while outliers exceeding 1 second is more common, thus increasing average response time. In fact, the average response time for the "Get booking" action is 1846 ms for 2000 users, significantly higher than for user numbers below 1500. We would not classify the application as unusable for 2000 users, but some users will experience rather long delays at some points in the process. It is reasonable to assume that adding even more users will significantly decrease application performance as we already is pushing the limits in terms of user experience with 2000 users.

The data for server response time supports the claims made in the the client-side RTT section above. This is expected as server response time, the time the server uses from a request comes in to a response is returned to the client, is a part of Client RTT. When the test hit 1500 users we see a spike in the average server response time in figure 8. This response time increases significantly also for 1750 and 2000 users.

The increase in response time for high user numbers is linked with queuing on the server. By looking at table 6 computed using Little's Law, we expect a request queue to form in the server when number of users exceeds 1500. As the average queue in the system grows longer, we expect the response time to increase. This is exactly what we have observed in our experimental tests. Queueing in the system is thus a contributing factor for the degrading response time performance of the application.

5.1.3 Analysis of the different steps in the operation

As we would expect, the steps in the operation that takes the most time, are those involving database interaction. The graphs in figure 6 and figure 7 display this relationship. The steps that involve database interaction are "Open search page", "Search", "Payment", "Load booking" and "Get booking". Looking at figure 6 and figure 7, we can clearly see that the average response times of these five steps are significantly higher than the average response times of "Select Room 1", "Select Room 2" and "Checkout".

Further, the average response times of the steps involving database interaction increase in line with number of users. On the other hand, the steps involving no database interaction have fairly constant response times, independent of the number of users.

There is also a significant variance of average response time within the group of steps that interact with the database. Generally, this variance may be explained by the difference in number of calls made to the database at each step, and the number of elements read from, or written to, the database. However, although "Get booking" and "Open search page" do the same, single call to the database, the average response time of "Get booking" is greater than the average response time of "Open search page", for every set of users. Thus, other factors must be taken into consideration in order to explain this difference.

Figure 6 and figure 7 suggests that "Search" is the most time consuming step of the booking operation, for every set of users up until 1250. This supports our assumption that the response time is correlated with the amount of database interaction. In order to load the search results, five database calls must be made (cf. section 3.1). Further, a large amount of elements may be loaded into the application, if the bookings table is densely populated. (cf. section 3.1). The "Payment" step computes free capacity in the same way as "Search". Thus "Payment" is also among the most time consuming steps. However, "Search" additionally loads and computes the prices of each room, for each date, and is thus slower than "Payment". The fastest of the database interacting steps is "Load booking". This is probably because "Load booking" only makes one call to the database, and that only 1 element may be returned.

There is less variance in the average response time of the steps involving no database interaction. More logical computation seems to affect the response time less than additional interaction with the database. Looking at figure 6 and 7 it may seem that "Select Room 1" has a slightly higher average response time than "Select Room 2" and "Checkout". This would make sense, since this step requires more computation than the two other (cf. section 3.2).

5.1.4 Error analysis

Under heavy load (> 1500 concurrent users) we recorded some errors. The most serious one was users able to book rooms when no free capacity was available. We recorded this by looking into the database after the tests. For some of the tests more than 100 bookings were made, even though only 100 rooms were available in the chosen time period. This is a serious error that needs fixing if the application were to be used in a production

environment. This error is likely to be caused by concurrent user requests accessing the database in quick succession and the database not updating free capacity quickly enough. Thus, two users can make a booking and just one room capacity being decremented. When this happens multiple times for one test, the number of total bookings exceeds 100, the maximum number allowed. The highest number of illegal bookings we recorded were 13, which is roughly 12% of all bookings made in the test.

5.2 Possible performance improvements

5.2.1 Connection pooling

Currently we use Hibernate's built-in connection pool mechanism. This is not recommended for production environments, which we are trying to replicate. From what we have read, Hibernate's connection pooling lacks features found in most proper connection pooling mechanisms. Whether such a change would have practical implications we do not know, but Hibernate claims that "The built-in Hibernate connection pool is in no way intended for production use." Thus, it would probably smart to change to a connection pooling package meant for production use, such as C3PO.

5.2.2 Database interaction

Based on our analysis, we believe that performance improvements best could be achieved by improving our data access layer. This is because database interaction seems to be the most costly part of the application logic. As pointed out in section 5.1.3, our tests showed that the average response time only increased for the steps that were dependent on reading from, or writing to, the database. Further, the response times were dependent on the amount of calls made, and the amount of data transferred. All other steps had fairly constant response times.

One step that certainly could be improved is "Search". At the moment, 5 calls are made to the database in order to load the correct search results. However, we should be able to achieve the same result by using a single, more complex query. For instance, we are now loading all the existing "room bookings" into our application in order to compute the correct free capacities. Checking for date-overlaps with the new booking is done in the application. Instead, our query should only ask for the number of rooms booked on the selected dates, for each room type. Had we done this, we would have received only five results (one for each room type) from the database. They way our application works now, we may in worst case receive hundreds of objects from the database. Additionally, each of these objects contains several fields. It is obvious that this is way more costly than transferring five aggregate counts. Since the "Payment" step confirms capacities in the same way as "Search" gets the capacities, the same improvement could be done here.

"Get booking", and "Open search page" does the same single call to the database. They both load all the hotels from the database, in order to show the correct choices in the search form/add extra room form. Each "Hotel" object has a list of associated

"Booking" objects which are non-lazily loaded into the application. This design choice was made, because having the booking objects for each hotel was a convenient way to access bookings in the manager/owner panel. However, this causes a lot of unnecessary overhead on the search and view booking pages, as the list of "Booking" objects are not used on the customer-side of the application.

Finally, it should be mentioned that these inefficient design choices did not affect the web application significantly when we did our tests. This is because the Tomcat server and database server were located on the same computer. Thus, transferring data between the application and the database caused minimal overhead, compared to what would be the case if these servers were located on two different computers. We would certainly want to reduce the amount of calls made to the database, and amount of data transferred, if these servers were physically separated. With the setup used now, improvements would probably only cause a minor reduction in the average response time. With the Tomcat server and Database server separated, the same improvements would probably cause a significant reduction in the average response time.

6 Conclusion

In this report we have analysed the performance of the Hotel Booking Web Application which we developed for assignment 2 in COMP9321. Our analysis found that the web application worked well up to around 1500 concurrent users. Somewhere between 1500 and 2000 users the average response time started to increase exponentially. Our tests also reviled that the application made some errors when the user load increased beyond 1500 concurrent users. The most serious error was that a few users were able to book rooms at a hotel with no free capacity. Further, we found that the bottleneck of our application is the data access layer. All the steps that required some database interaction had average response times greater than all steps that did not interact with the database. Based on this, we suggested that in order to improve the performance of the application, we should focus on improving the data access layer. One of the improvements we proposed, was reducing the amount of transaction involved in computing the free room capacities. By making this procedure more efficient, we could possibly reduce the amount of double-bookings.