18-749 – Team 5 Evaluation Results

Team Members:

Patty Pun
Kevin Smith
Zhang Yi
Felix Tze-Shun Yip

Introduction

The purpose of this evaluation was to determine overall system performance for our fault-tolerant baseline application using end-to-end latency as the performance metric. The analysis required that we run our system in each of the 48 combinations possible using the following:

- 0, 20, 40 ms inter-request time (time between client requests)
- 4, 256, 512, 1025 byte replies
- 1, 4, 7, 10 clients

For each configuration, clients would make 10,000 requests, and data would be recorded at both clients and servers. For each request, the client would record the time the request went out, the time it returned, and the name of the request being made. The server would record the time the request arrived, the time it left, the name of the request, and the name of the client making the request.

To allow for this data collection we had to choose those client requests which we would test and modify them in some way. The details of this modification can be found in the next section. We also had to add command line argument parsing to the client so that it could receive the different parameters of evaluation (inter-request time, reply size, number of clients).

After making these changes, we could begin data collection. For data collection, we made use of the Linux games cluster in CMU's ECE department. These 15 machines are each running dual 2.8 GHz Pentium 4 processors with 1MB on-chip cache. A single bash (http://www.gnu.org/software/bash/) script was created which ran through all 48 configurations. For each configuration, the script would remotely start servers and clients on machines in the cluster. It would then wait for all clients to finish, and after doing so, would stop the servers and move on to the next configuration.

Using these results, we could then measure several properties of our system including latency, server request rate, and server throughput. In addition, we could breakdown the latency measurement into middleware and server components. After collecting the data and making these calculations, five different types of graphs were to be created:

- Line plots of latency for increasing number of clients and different reply sizes (no pause)
- Area plots of (mean, max) latency and (mean, 99%) latency, sorted by increasing mean values
- Bar graphs of latency component break-down for outliers and normal requests
- 3D scatter plots of reply size and request rate impact on max and 99% latency
- Latency vs. throughput

Each of these graphs would present us with a unique view of the latency seen in our system. The next section of this document will discuss the changes required for the client invocations. Following that section, results are presented in the form of the five graph types listed above. Finally, the document concludes with an analysis of the results.

Client Invocations

METHOD	***ONE_WAY***	***DB_ACCESS***	***SZ_REQUEST***	***SZ_REPLY***
`login()`	No	Yes	? bytes ¥	4, 256, 512, or 1024
`logout()`	No	Yes	? bytes §	4, 256, 512, or 1024

¥ 24 bytes + (2 bytes * length of clientname)
§ 18 bytes + (2 bytes * length of clientname)

public int [] login(String username, String password, String clientname, int replySize)

In the baseline version, the login() method receives two strings as parameters, the username and the password (In Java, strings use unicode characters which are 2 bytes). It queries the database to see if the password is correct, and if so, updates the user table to specify that the user is logged in. The login() method returns a 4 byte acknowledgement in the form of an int. This acknowledgement tells the client whether or not the login action was successful. The acknowledgement is false if the method could not connect to the database or the password is incorrect.

In the evaluation version, the login() method receives two additional parameters, the connecting client machine name and a reply size. The login() method performs the same functionality as the baseline version but also adds the time it started, the time it ended, the connecting client machine name, and its method name to the four corresponding probe lists. Timestamps are recorded as the number of microseconds since the application started. We had to modify our system to use Java 1.5 as this newest version of Java (http://java.sun.com/) provides a method for measuring time in nano seconds. After receiving notification from all clients that they have finished, the server will write all its probe data to disk. In addition to these changes, the method returns an int array where the first value is the acknowledgement and whose size is specified by the replySize parameter divided by 4 (in Java, int primitives are 4 bytes).

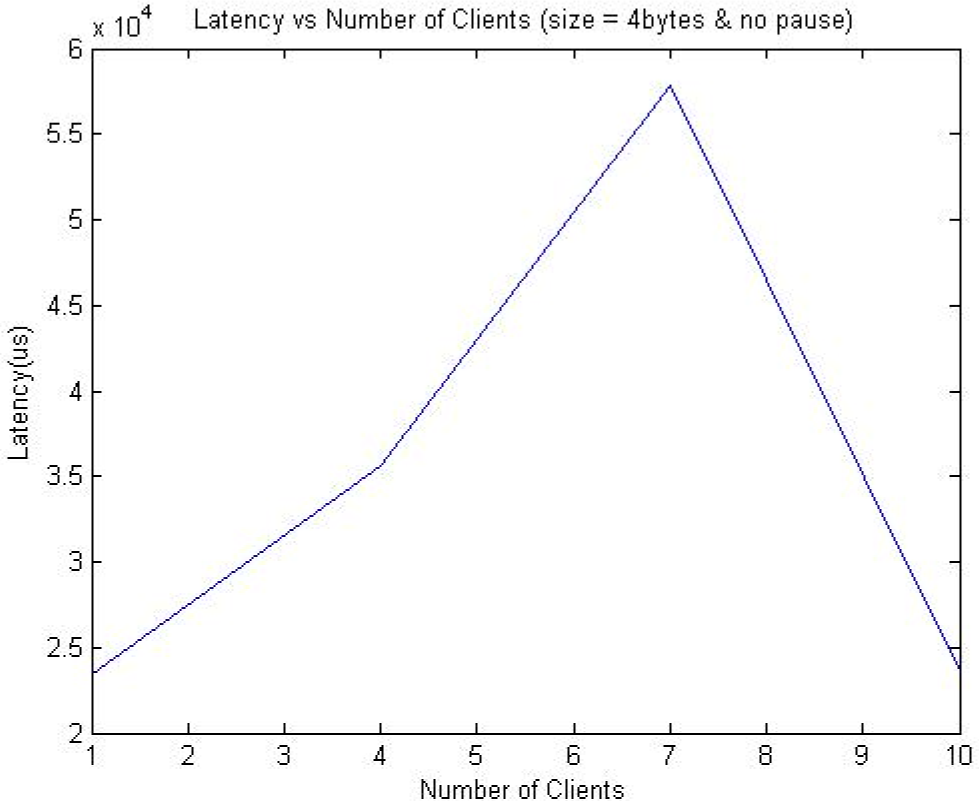
public int [] logout(String username, int sessionNumber, String clientName, int replySize)

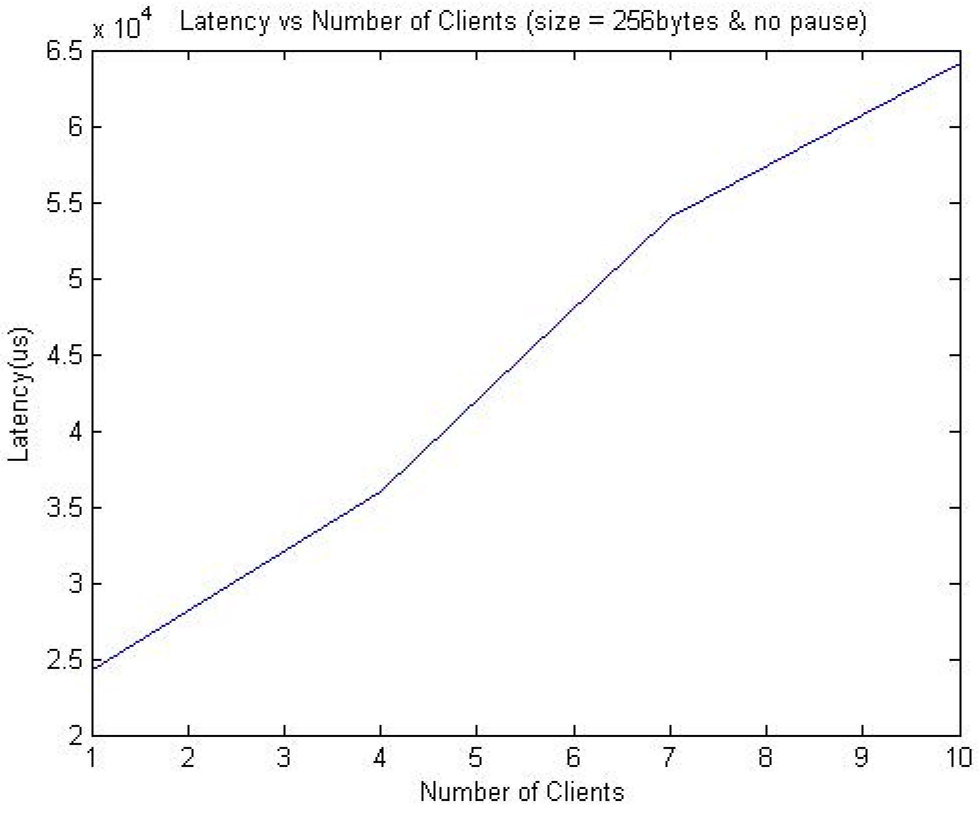
In the baseline version, the logout() method receives a username and a session number. The session number specifies the chat room in which the user was chatting. The logout() method sets the logged_in value to false for the user's entry in the users table in the database. It then looks at the messages table in the database, and if there are no longer any users in the room specified by sessionNumber, it clears the messages table of the messages from that chat room.

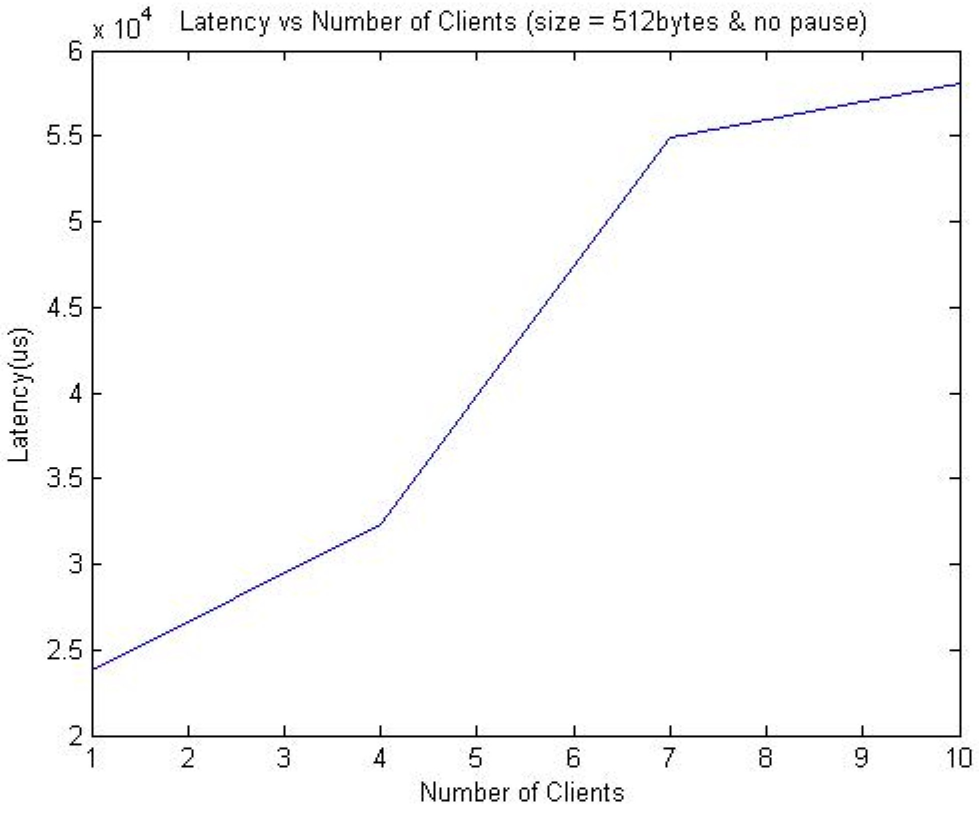
In the evaluation version, the logout() method has the same changes and additions as specified above in the description of the evaluation version of login().

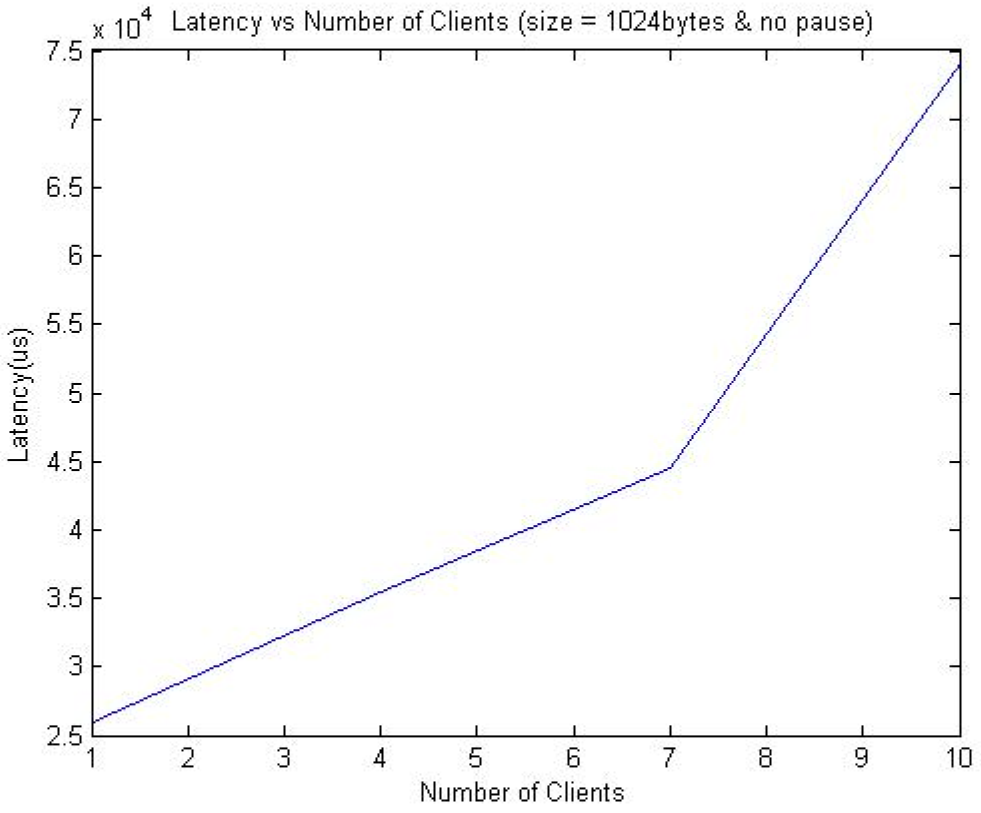
Graphs

Line Plots of Latency



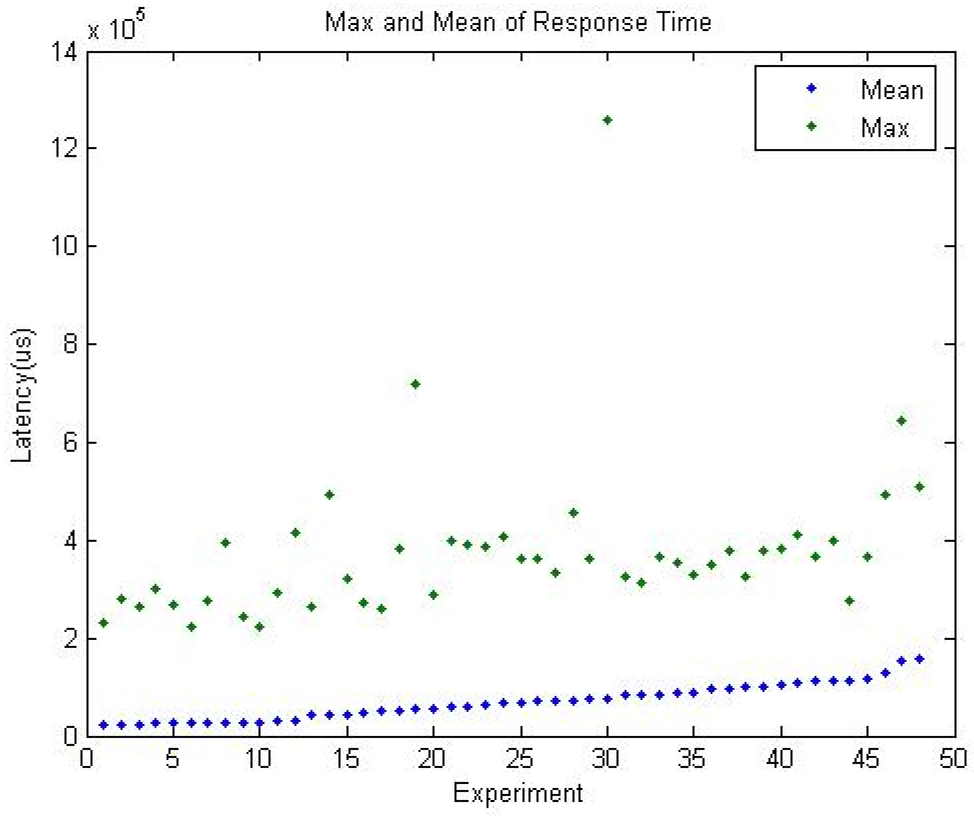

Latency vs Number of Clients (size = 256bytes & no pause)
x 10^4
Latency(us)
Number of Clients
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1
2
3
4
5
6
7
8
9
10


Latency vs Number of Clients (size = 512bytes & no pause)
x 10^4
Latency(us)
Number of Clients
6
5.5
5
4.5
4
3.5
3
2.5
2
1
2
3
4
5
6
7
8
9
10

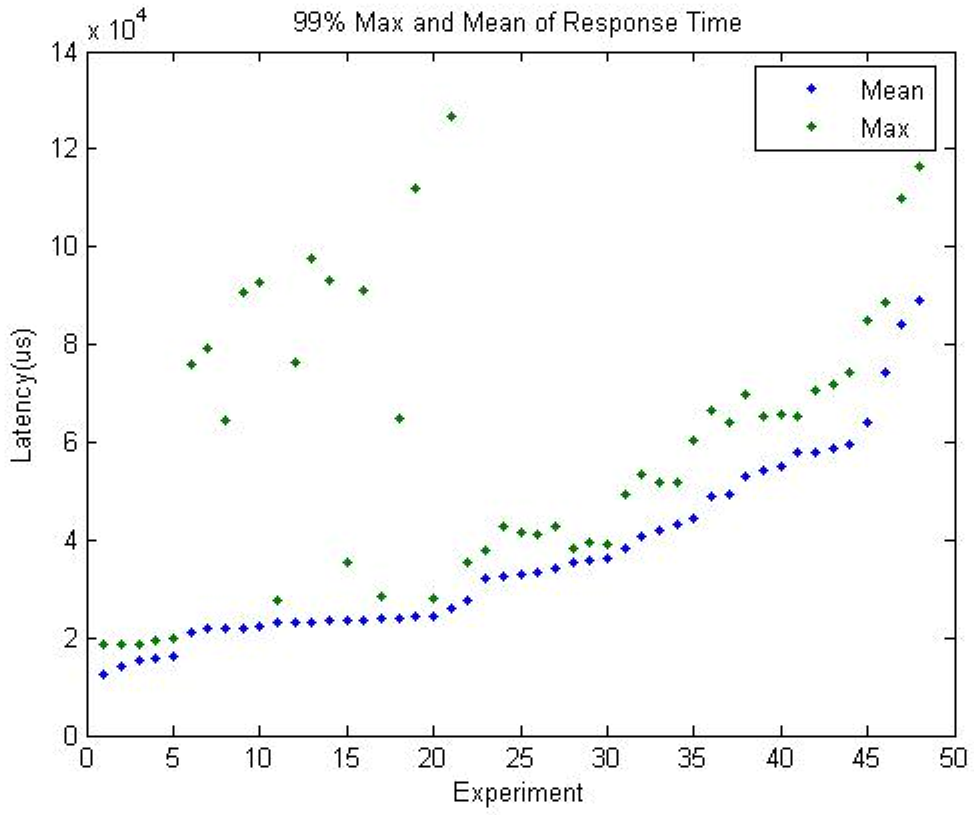

Latency vs Number of Clients (size = 1024bytes & no pause)
x 10^4
7.5
7
6.5
6
5.5
5
4.5
4
3.5
3
2.5
Latency(us)
1
2
3
4
5
6
7
8
9
10
Number of Clients

Area Plots of Latency

Maximum Latency

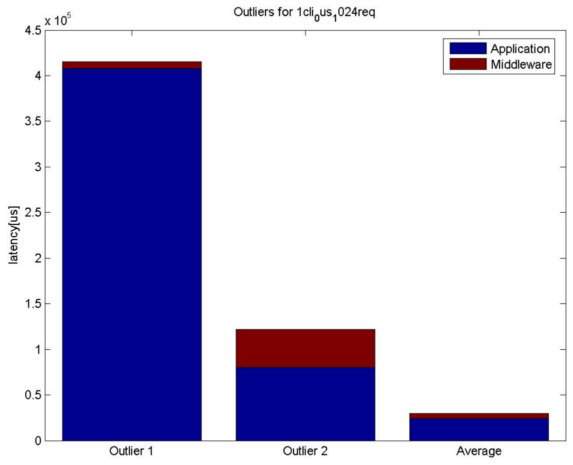


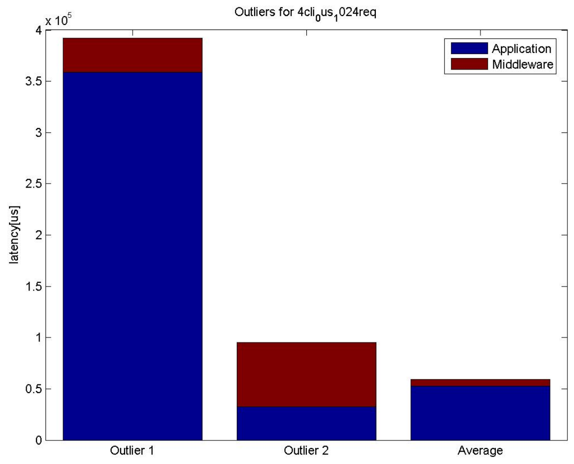
99% Latency

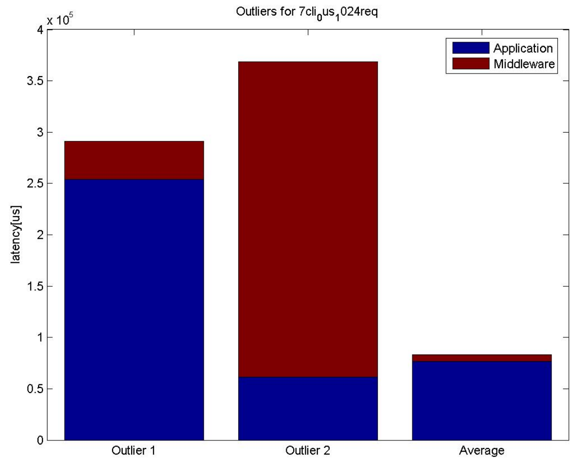

99% Max and Mean of Response Time
x 10^4
Latency(us)
Experiment
Mean
Max
0
2
4
6
8
10
12
14
0
5
10
15
20
25
30
35
40
45
50

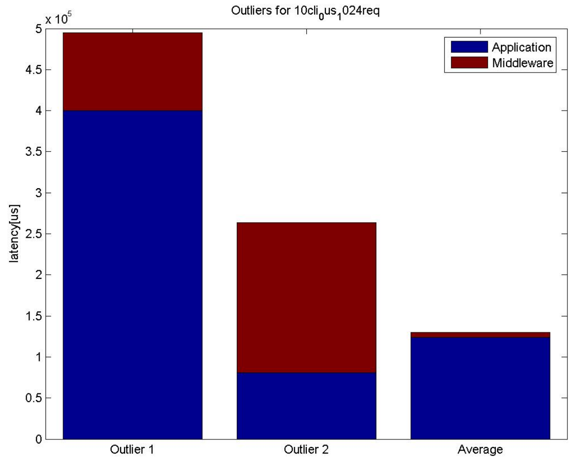
Bar Graphs of Latency Component Break-Down

Increasing Number of Clients (0 ms inter-request time, 1024 byte reply)

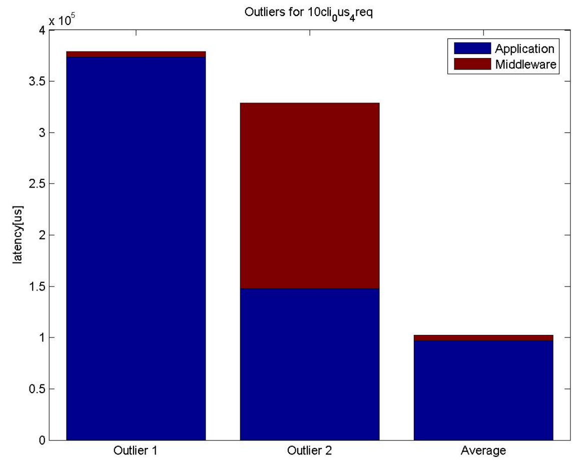


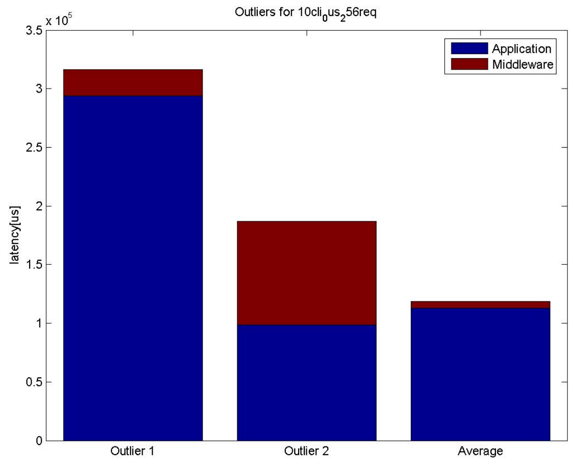


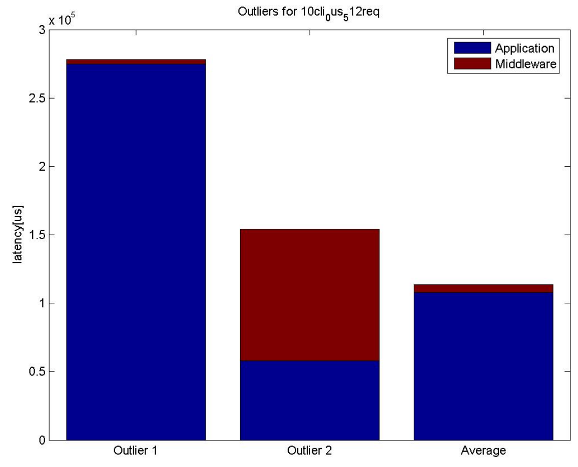


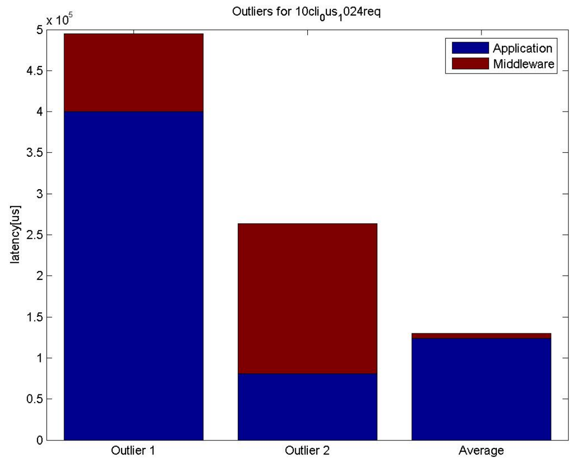


Increasing Reply Size (0 ms inter-request time, 10 clients)



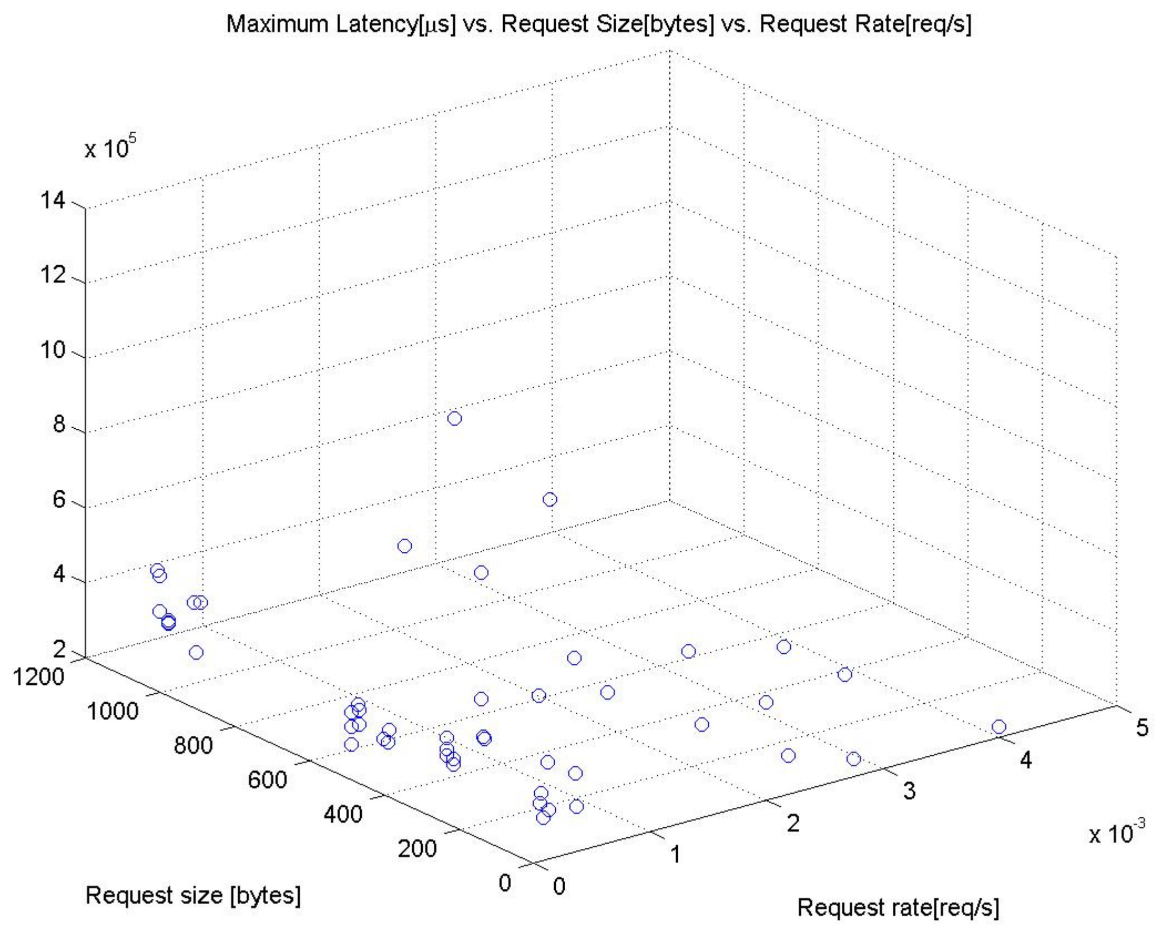




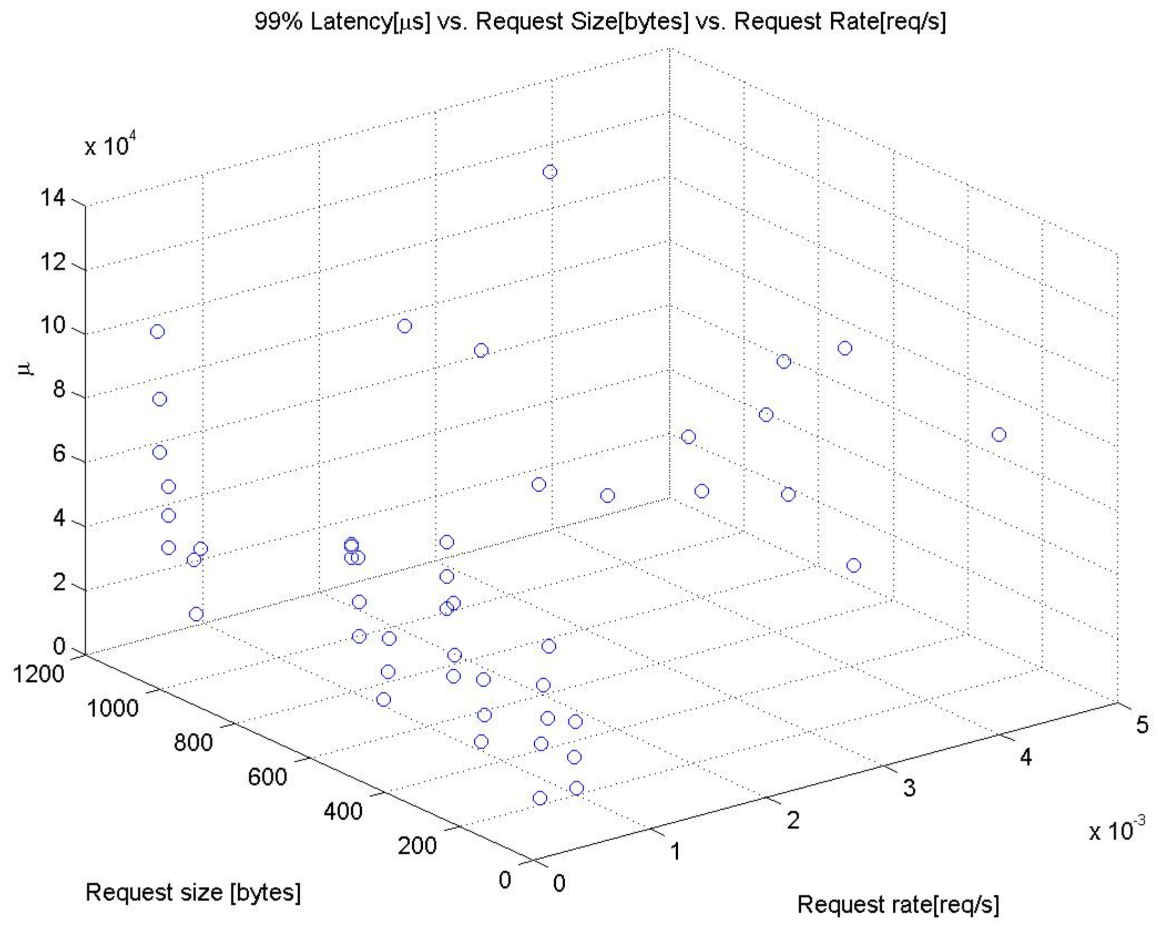


3D Scatter Plots

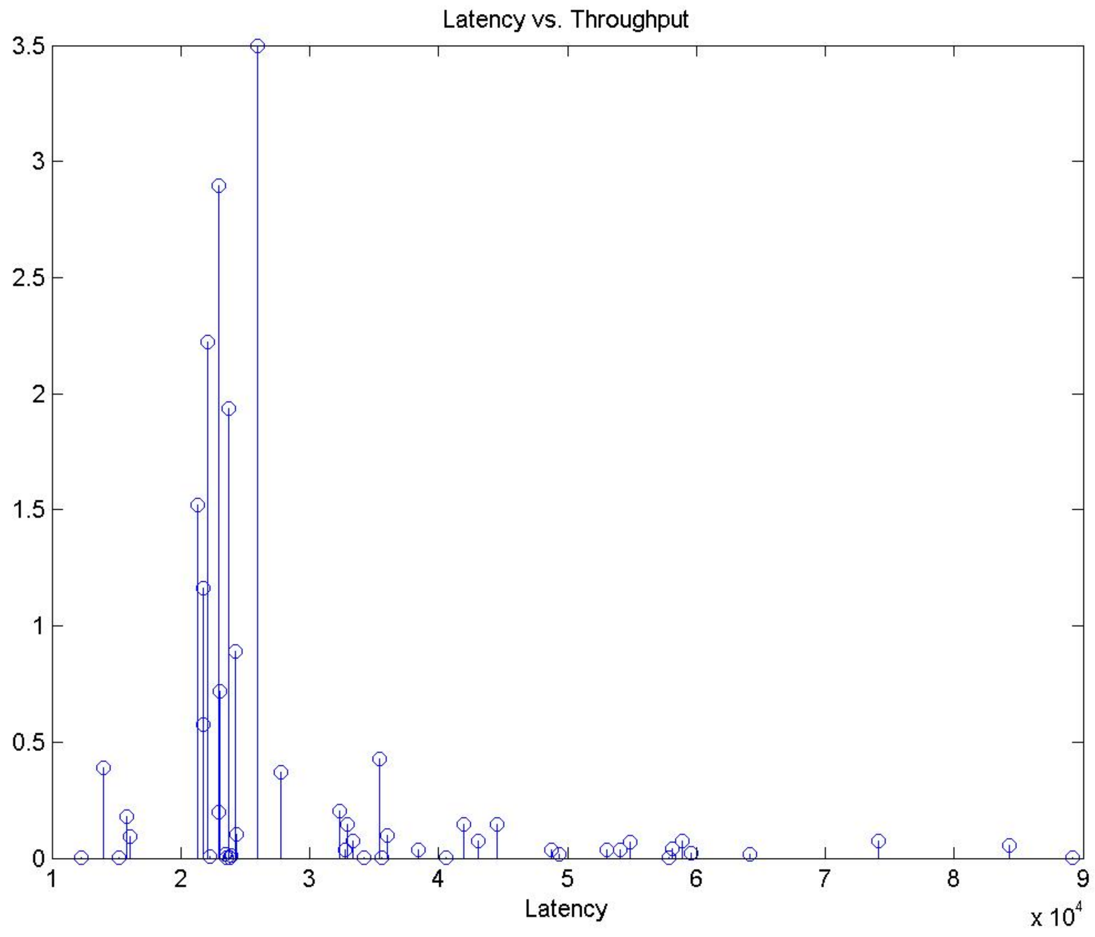
Maximum Latency



99% Latency


99% Latency[μs] vs. Request Size[bytes] vs. Request Rate[req/s]
x 10^4
14
12
10
8
6
4
2
0
μ
1200
1000
800
600
400
200
0
Request size [bytes]
0
1
2
3
4
5
x 10^-3
Request rate[req/s]

Latency vs Throughput



Discussion

An analysis of the system from the experiment reveals predictable performance tradeoffs as the number of clients and reply size increases. With the exception of one possible errorenous result in our configuration with 4 byte replies and 0 ms inter-request time, in which a large number of clients actually lowered the latency, the rules are generally predictable.

It was a surprise to see that the testing client, which worked with other configurations yielded different results. There were a number of unexpected outcomes during the analysis of the system including the difference in graph shapes between responses of 256 bytes and 512 bytes versus 1024 bytes. We would expect a similar pattern occuring since uniform configurations exist at all levels (Middleware, Operating System and Hardware).

Differences may be attributed to lower-level issues such as cache size and CPU optimization. However, it is likely that the examined section may be the earlier sections of the graphs observed in the graphs with 4, 256 and 512 bytes of data returned per function call. This theory may be tested with a by using larger reply sizes that relates to multiples of the RAM size of the running processor. In addition, different systems with different RAM sizes may also be used to verify this pattern.

The performance of the system is currently relatively poor since the latency is relatively high for the number of clients. With reference to a point that appears similar through the many results, with 7 clients and nearly 6 x 10^4 us delay per function call which translates to 0.06 seconds meaning only 16 times may be called per second of execution.

It may also be said that the middleware component of latency increases with increasing numbers of clients as well as with increasing reply size. This can be seen in the bar graphs which present the breakdown of latency into its application and middleware components.

As for the Magic 1% phenomena, our system does not appear to adhere to this rule. This conclusion is hard to make though considering the relatively small size of our data.