Advanced Systems Lab Report

Autumn Semester 2019

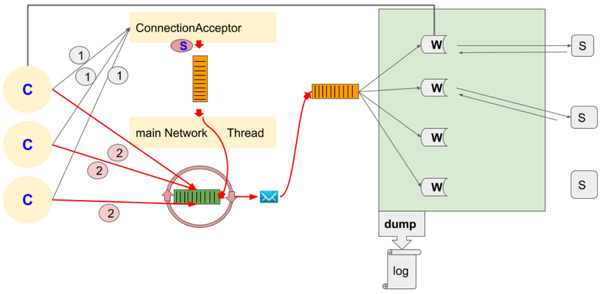
Name: Anastasiia Ruzhanskaia
Legi: 18-904-029

December 16, 2019

1 System Overview (80 pts)

1.1 System Description (20 pts)

The middleware scheme is presented on the following picture:



The middleware has a network thread - Middleware.java, which accepts messages and puts them into the task queue. It has a helper thread - ConnectionAcceptor, which accepts new connections. After accepting they are put into a shared queue (one of the orane queue on the scheme) between Middleware and ConnectionAcceptor thread. This queue is then used to fill the array of connections across which Middleware iterates.

Middleware always check if there is something in this queue and if there is a new connections, it puts it into the array of connections (green array on the scheme). We need to have also a separate array, as we would like to delete connections, which are not active.So while iterating across array, we are able to add new connections there and delete unnecessary ones.

Middleware traverse the array of connections one by one (denoted like a circle around green array on the scheme), checking if there is something to read from the corresponding socket. If there is, we read while there is something to read. If we read until a new line, then we put this request inside a queue. This way we guarantee the correct processing of GET requests.

Summarizing, maximum 2 threads at a time manage connections. The ConnectionAcceptor thread is sleeping mostly, it works only at the beginning of each experiment.

Next class is WorkerThread.java. It shares TaskQueue with Middleware thread. For both queues - Connections queue and Tasks queue Java LinkedBlockingQueue class is used. It has an advantage over other Java class queue implementations, it has a blocking method take(), which worker threads need. So each thread is blocked on the queue when there are no messages inside. This actually allows also the worker to get equal amount of work.

There is no fixed worker threads pool. The array of Worker threads is newly allocated when the worker configuration changes.

The instrumentation was added to the WorkerThread thread and it measures following parameters: the number of requests left queue, the number of successfully completed requests, queue size, time in queue, processing time of the middleware (this includes waiting from server), time spent just in parsing and sending, time the request spent just in server, overall request time in middleware - middleware response time.

The places of measuring different things are shown on the picture above.

As additional instrumentation, dstat and qperf are used. Dstat - for CPU, memory statistics, qperf - for bandwidth and latency. Unlike ping, qperf operates with tcp protocol and provides more precise results, comparing to ping, which operates with ICMP protocol.

1.2 Middleware Data-Structures (20 pts)

Internally two main queues are used: one for adding new connections (Queue 1) and one for adding new tasks (Queue 2).

Queue 1 consists of Java Sockets. Queue 2 consists of types QueueStructure.

```
        public class QueueStructure {
                String request;
                Socket connection;
                Instant enqueueTime;
        }
```

As mentioned is section above, both queues are implemented using Java LinkedBlocking queue.

Network thread is represented through the Middleware class and has one helper class - ConnectionAcceptor thread, they were described in previous sections.

Request are stored in a task queue, then each Worker thread takes them and stores in local memory until the answer on this request is not sent back to server.

Code instrumentation is done using Java Instant class. In every part of the program, when the time measuring should end or start (see picture in previous section), call Instant.now() is inserted, in the end, if the request was successful, difference between recorded timestamps are computed.

One timestamp is passed across the queue - enqueue time, we can't track this from th worker prospective, so the Middleware thread records enqueue time and then put the QueueStructure data structure inside the queue. Then worker use this value to find difference between it and dequeue time, for example.

For the code instrumentation special data structure is used - WorkerStats. This data structure is copied inside StatsPrinter thread every 5 seconds. The fact that we copy them before printing allows each worker thread to continue executing and updating WorkerStats without sharing the statistics for variables during whole printing process. The separate thread for printing allows workers not to conquer for a one common resource, which will be an issue if every worker printed by itself to a common file.

StatsPrinter copies the statistics from workers in a loop, then it print the stats, using it's local refreshed variables. It stores always the statistics data from previous 5 seconds and then finds the difference between two statistics during 5 seconds. This result is precise, because statistics are recorded by worker thread per request, which means that only when the request is successfully sent, all statistics is updated in a synchronized fashion. We need synchronized keyword over the updates because 5 seconds trigger or printing might happen somewhere inside updating process and then the statistics will be inconsistent.

After finding the difference between two statistics values, StatsPrinter dumps results for each worker thread in one file in a loop. This way we receive a bunch of lines in log file (number of lines is equal to the number of workers) each 5 seconds.

One detail to be mentioned is that statistics numbers are not zeroed after the printing and overall during the execution. We will need for this additional synchronization. We use the fact that Java does not generate exceptions when the overflow happens, so despite the fact that statistics numbers by themselves will be incorrect, their difference will be always correct and we don't need to think about zeroing the values after a dump.

The WorkerStats structure consists of the following fields (each field is explained on the line before):

```java
class WorkerStats implements Cloneable {
        //thread number
        int worker;
        // requests, that returned from the server with positive answer
        int successfulRequests;
        // accumulated time, spent by reqeusts in queue
        long timeInQueue;
        // accumulated sizes of queue after each dequeing
        long sizeOfQueue;
        // requests, for which we are counting this time
        long requestsLeftQueue;
        // accumulated service processing (server + sending) time for successful requests
        long timeInServer;
        // accumulated processin time between getting from queue ans sending to server
        long timeInParseAndSend;
        // accumulated overall response time in middleware
        long timeToProcessRequest;
        // accumulated time to process request plus queue time
        long timeToProcessRequestAndQueueTime;
        // accumulated cache misses
        int cacheMisses;
        // accumulated error strings
        StringBuilder errors;
        // accumulated number of requests, which were sent to each server
        int[] requestsPerServer;
        int nServers = 0;
        StringBuilder errorString;
        ...
}
```

When there are no requests, StatsPrinter thread still continues to dump statistics, but of course this will be just zeros. However, this behavior means that statistics collection is completely decoupled from the WorkerThread logic and from Middleware logic. Also, the whole Middleware does not implement additional logic for tracking experiments.

Describe the data-structures used internally in the middleware to manage messages and connections. Explain how you implement/design network threads, queues, request storage, code instrumentation, performance measurement storage, etc.

1.3 Middleware Request Handling (16 pts)

1.3.1 Request Parsing (8 pts)

Middleware thread just put requests in the queue. Then the Worker thread takes them, uses Java function for Strings in order to parse them - split() function. With this function it is possible to check two main conditions - that the first word is get and that if it is get, then the only word afterwards could be the key. Only if this holds, we process further, if not, we return back to the server "END" string and record the error.

1.3.2 Requests Processing (8 pts)

1.4 Work Balancing (8 pts)

Work balancing is done in round robin fashion. Each worker stores a current index of the server he wants to send message to. After each processed request this index is increased by one modulo number of servers.

In the obtained log files we see that the number of requests sent to each server is equal. They also should be equal by design as we process synchronously and each request change the server index. The logs related to this can be seen in mw.log file (columns 9, 10, 11 for three servers).

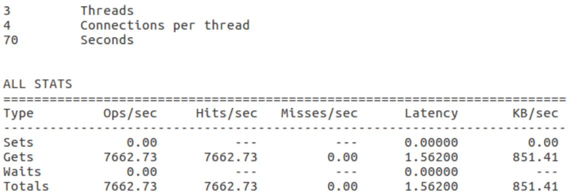
1.5 Data Processing (8 pts)

Each memtier instance and each middleware are printing to separate log files. All in all we have 6 experiments.

For clients each experiment is put in a separate folder: part1, part2, part3 and etc. Each folder contains client files, for example, client1.log, client2.log and etc, one for each memtier instance during this experiment. Middleware always dumps into the same log file.

Processing is done with Python, using numpy and pandas libraries. The last one allows to group values very conveniently, find mean() and sum() of the values in a column.

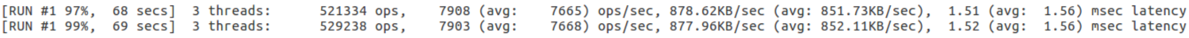
Client logs For clients I don't need to dump something additional, all statistics is provided by memtier. It is dumped in this format:



```
3         Threads
4         Connections per thread
70        Seconds


ALL STATS
========================================================================
Type        Ops/sec     Hits/sec   Misses/sec      Latency       KB/sec
------------------------------------------------------------------------
Sets           0.00          ---          ---      0.00000         0.00
Gets        7662.73      7662.73         0.00      1.56200       851.41
Waits          0.00          ---          ---      0.00000          ---
Totals      7662.73      7662.73         0.00      1.56200       851.41
```

Also it dumps every second some additional intermediate results:



```
[RUN #1 97%,  68 secs]  3 threads:      521334 ops,    7908 (avg:    7665) ops/sec, 878.62KB/sec (avg: 851.73KB/sec),  1.51 (avg:  1.56) msec latency
[RUN #1 99%,  69 secs]  3 threads:      529238 ops,    7903 (avg:    7668) ops/sec, 877.96KB/sec (avg: 852.11KB/sec),  1.52 (avg:  1.56) msec latency
```

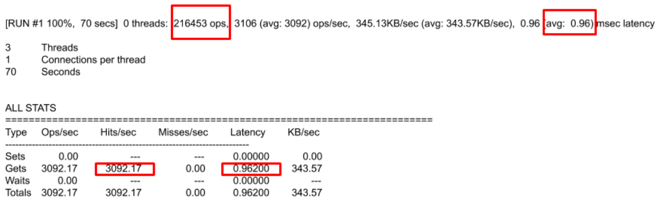
In my data analysis for clients I operate with the reported Throughput (Hits) in ops/sec, Throughput in KB/sec, Response time and Overall number of requests processed (from intermediate printing).

Obtain final results for clients plots $Throughput = sum(throughput_client_i)$ across all memtier instances. With the response in theory we could take just the average but it is not correct if the client numbers varies more than on 10 percents. This could happen if the network conditions for different client machines are different. According to the conducted experiments with qperf, the latency numbers for client machines to servers differ sometimes in significantly, which leads to the difference in obtained throughput and response time. Interactive law is correct for all these numbers.

In order to find the right average, the way the memtier prints statistics was investigated. For response time it collects the response time from all it's threads and divides them on overall number of the requests in all threads. The latter is printed in intermediate memtier logs. We will use it in order to find the sum of response times across all threads of one memtier instance.

Then we will sum up these times across all memtier instances as well as the number of requests, which we receive from additional statistics. These two global sums will be divided on each other and the average response time will be received.

Picture which shows which information is used for logs:



```
[RUN #1 100%,  70 secs]  0 threads: 216453 ops,  3106 (avg: 3092) ops/sec,  345.13KB/sec (avg: 343.57KB/sec),  0.96 (avg:  0.96) msec latency

3         Threads
1         Connections per thread
70        Seconds


ALL STATS
========================================================================
Type     Ops/sec      Hits/sec    Misses/sec      Latency       KB/sec
------------------------------------------------------------------------
Sets        0.00          ---           ---       0.00000         0.00
Gets     3092.17      3092.17          0.00       0.96200       343.57
Waits       0.00          ---           ---       0.00000          ---
Totals   3092.17      3092.17          0.00       0.96200       343.57
```

We could write this process the following way:

```
sum_time = 0
sum_requests = 0
for all memtier instances:
        sum_time += response_time_per_memtier * sum_requests_per_memtier
        sum_requests += sum_requests_per_memtier
average_response_time = sum_time / sum_requests
```

Middleware data analysis On the middleware data is accumulated in 5 seconds window per worker thread and then is collected by the printing thread. The data for each measured parameter is printed in a separate column.

Then these columns are read as .csv file and processed with Python pandas library.

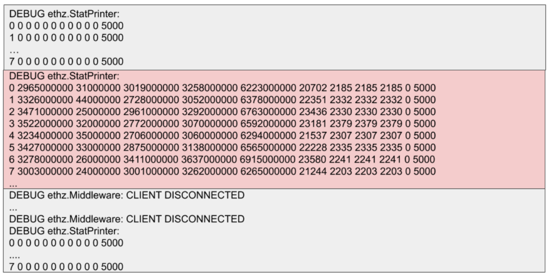
Differentiation between different experiment in middleware log file As the logs are dumped continuously just with StatsPrinter thread, there is no explicit borders between experiments in middleware log file. In my scripts I always insert 10 seconds sleep command in order to:

- make sure that unnecessary client will be disconnected;
- to differentiate different experiments;
- to make sure all processes from current experiment finish by the next experiment.

This way in general (there are couple of other conditions that needs to be checked against what follows these chunks) when I find chunks of 8, 32, 64 zero lines statistics, I can say that this is the border between two experiments, because during this 10 seconds sleep window middleware will be able to print zero statistics at least once.

The middleware logs are presented on the following picture:

The parsing script separates one big middleware file on N small .log files, each for one experiment (one value size, one umber of workers, one number of clients).



```
DEBUG ethz.StatPrinter:
0 0 0 0 0 0 0 0 0 0 0 5000
1 0 0 0 0 0 0 0 0 0 0 5000
...
7 0 0 0 0 0 0 0 0 0 0 5000
DEBUG ethz.StatPrinter:
0 2965000000 31000000 3019000000 3258000000 6223000000 20702 2185 2185 2185 0 5000
1 3326000000 44000000 2728000000 3052000000 6378000000 22351 2332 2332 2332 0 5000
2 3471000000 25000000 2961000000 3292000000 6763000000 23436 2330 2330 2330 0 5000
3 3522000000 32000000 2772000000 3070000000 6592000000 23181 2379 2379 2379 0 5000
4 3234000000 35000000 2706000000 3060000000 6294000000 21537 2307 2307 2307 0 5000
5 3427000000 33000000 2875000000 3138000000 6565000000 22228 2335 2335 2335 0 5000
6 3278000000 26000000 3411000000 3637000000 6915000000 23580 2241 2241 2241 0 5000
7 3003000000 24000000 3001000000 3262000000 6265000000 21244 2203 2203 2203 0 5000
...
DEBUG ethz.Middleware: CLIENT DISCONNECTED
...
DEBUG ethz.Middleware: CLIENT DISCONNECTED
DEBUG ethz.StatPrinter:
0 0 0 0 0 0 0 0 0 0 0 5000
....
7 0 0 0 0 0 0 0 0 0 0 5000
```

1.6 Experimental Setup (8 pts)

For running the whole set of experiments some initial manual configuration of the machines is required. 1. Launch tmux sessions on each of the server machines, launch memcached there. 2. Connect to the first client (Client1 machine in setup) and launch there a tmux session. Then run a script for all experiments.

Further everything is automated. If additional instrumentation like dstat should be run in parallel, this should be done manually.

Scripts for each out of 6 experiments (description further) are run in a loop over values sizes, so that population of the database happens only 4 times. Each populations takes 1000 seconds. In all experiments the miss rate is 0.

Inside the outer loop over value sizes, there are 6 experiments for a fixed value size (further one line on our graphs):

- Population of the database of all three servers;
- 3 clients - 1 server machines experiment;
- 3 clients - 3 servers machines experiment;
- 3 clients - 1 middleware - 1 server machines experiment;
- 3 clients - 1 middleware - 3 servers machines experiment;
- 3 clients - 2 middleware - 1 server machines experiment;
- 3 clients - 2 middleware - 3 servers machines experiment.

And this bunch of experiments repeats four times for each value size. Each experiment inside looks like a nested loop. In case experiments without middleware, this a loop across clients and repetitions. In case experiments with middleware this a loop at first over the worker threads, then over the client number and then over the repetitions.

All scripts are launched from the one client machine. It is able to connect all other machines, as they are all populated with a public key for a ssh connection. Client1 machine launches a memtier instance on every other client machine and also starts and stops the middleware when the worker numbers is changed. This way of running allows to run all the clients almost simultaneously which is then proved by the experimental results. In particular, this is due to the fact that ssh latencies are a lot smaller when connection from internal network of the cloud infrastructure than from local computer.

1.7 Additional Remarks (optional, remove if not needed)

There are couple of details, which are not implemented or implemented in specific way in middleware. Firstly, when middleware finishes, we rely on Java machine to close all sockets connections.

Secondly, in the logs, currently middleware prints with the time printed before the bunch of log lines. This time is not parsed in the log processing script, so it is necessary to delete this times in front of each bunch manually. This could be done in one second in Vim by deleting everything corresponding to the time pattern.

2 Baseline without Middleware (120 pts)

The plot trends and saturation points description is included either in Result analysis or in Explanation section.

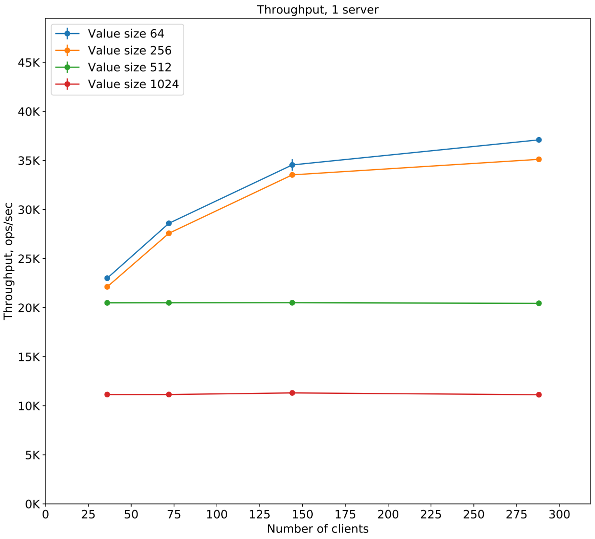
2.1 One Server (50 pts)

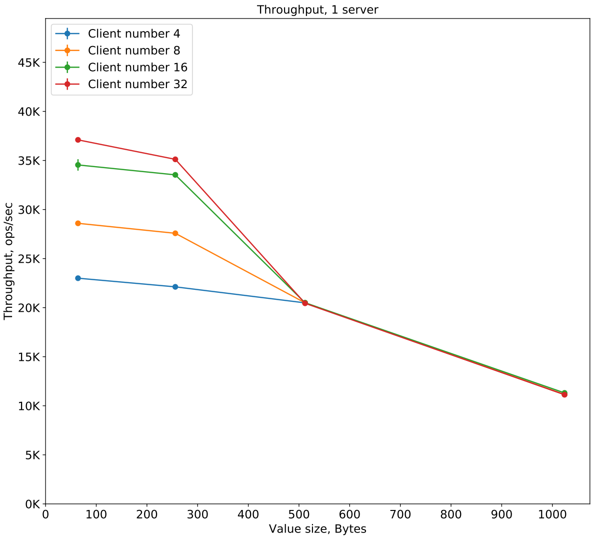
2.1.1 Setup (5 pts)

The same deployment was used in all experiments in the report. There was a global loop over value sizes and then local nested loops across client numbers and repetitions. For one and three servers the setup was almost the same. Here will be explanation for both. For each value size of global loop was run a nested loop: at first over client numbers, the over 3 repetitions. In each loop iteration the first client makes three connections (to the other two clients and to itself) and launches script experiment.sh, passing there necessary parameters. For the configuration with 3 servers first client makes 9 connections and launches script experiment.sh, passing there necessary parameters.

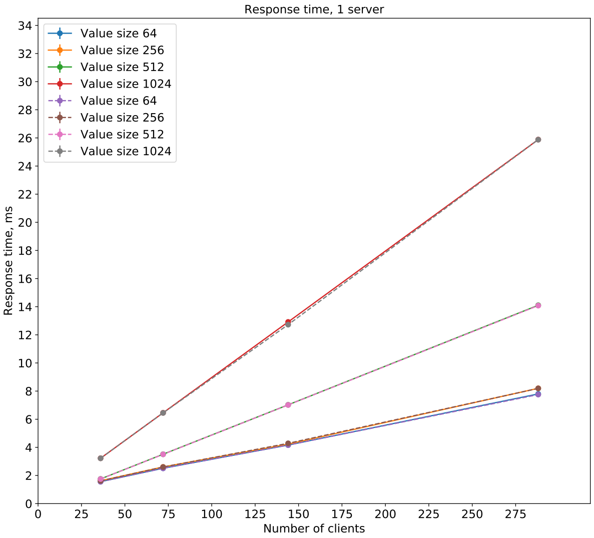
In experiment.sh script one memtier instance is launched. Each memtier instance connects to the same memcached instance (or to the one of three memcached instances). Each iteration of the loops first client waits until processes on all machines finish, waits additionally 10s and then runs next experiment iteration.

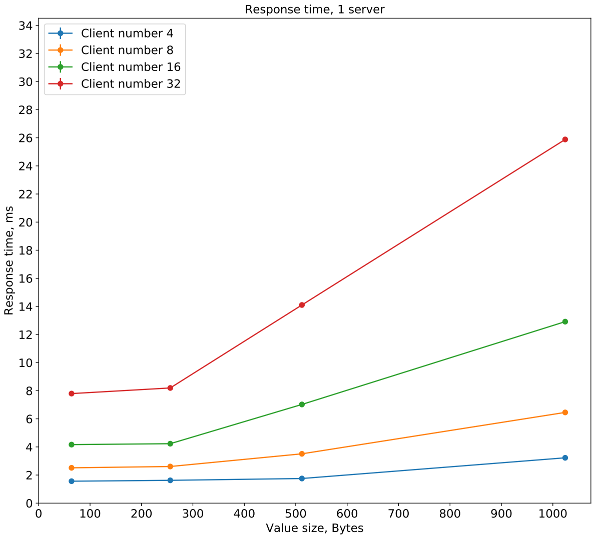
2.1.2 Throughput (10 pts)





2.1.3 Response Time (10 pts)

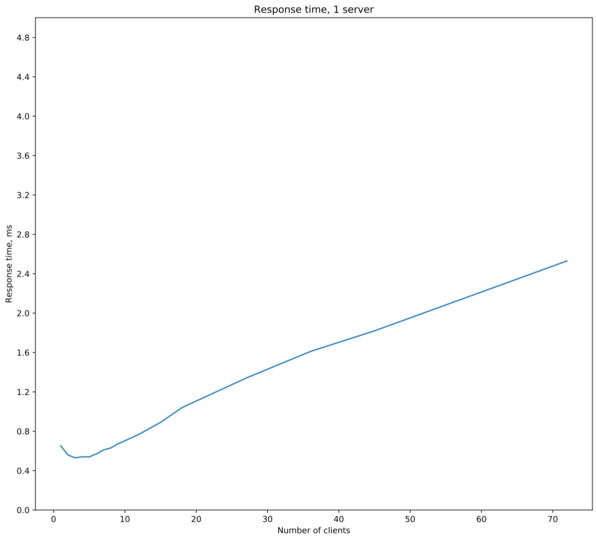




2.1.4 Result Analysis (5 pts)

The results plotted on the response time and throughput graphs are consistent, the interactive law holds and is illustrated on one of the response time figures. However, we see a linear grow for the response time on the plots with clients numbers on the X axis. To analyze this we will first plot the plot Response time - Throughput 2.1.5, where we clearly see that from the very beginning the line for even the first client number is increasing, which means that we are partially saturated. Looking on the throughput graph, we can also observe not a 2x grow from the very beginning with the growth of the clients in two times.

Additional experiment was conducted in order two show the place, where the response time starts transiting from the steady state to an increasing state.



We see that already around 12 clients the response time start to grow (before it has values always around 0.5 - 0.65 ms).

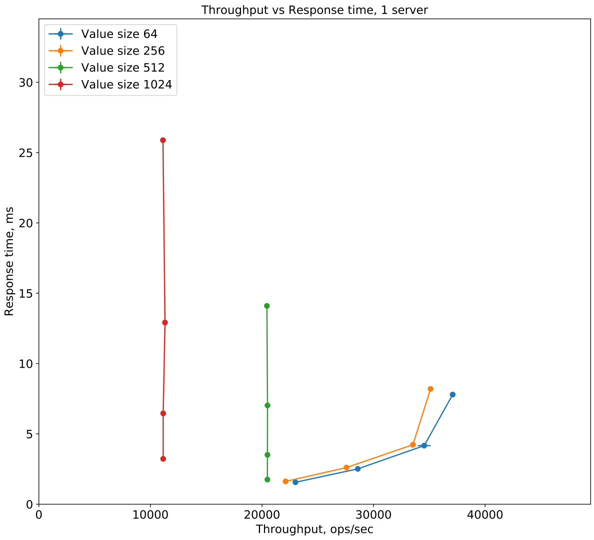
Thus, we could confirm that the graphs are correct, the interactive law holds, but initially the system is in partially saturated phase, this is why we don't see a dramatic increase in the throughput while increasing the number of clients and also see the grow in response time numbers from the very beginning.

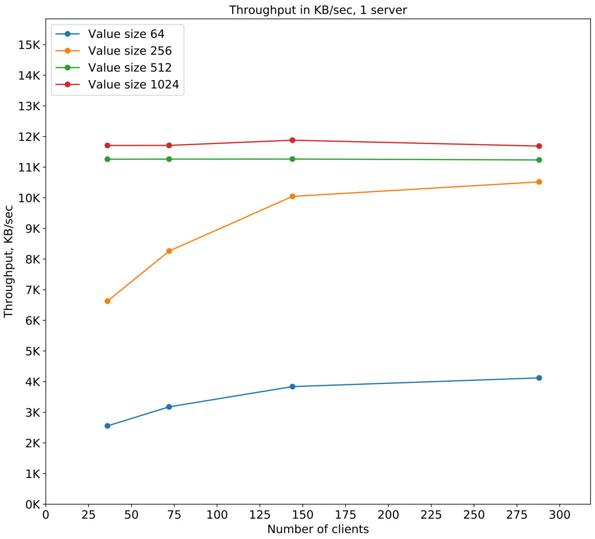
2.1.5 Explanation (20 pts)

Looking on throughput graph 2.1.2 we see the complete saturation for the value sizes 512 and 1024 and partially saturated (undersaturated) phases of the other two value sizes. With the increase of clients the throughput for value sizes 512 and 1024 stays the same on the level of about 20K ops/sec and 10K ops/sec operations correspondingly.

Along with that, we also mentioned that the interactive law holds, and the response time is constantly growing with the throughput for the value sizes 64 and 256, both with the increase of client numbers and value size. For the two value sizes, where we see complete saturation, the response time plot has, as expected (according to the interactive law), a rather steep slope, it grows fast both with increase of the number of clients and value size.

To analyze the difference here we could address to server utilization and to the bandwidth limitation in our network. We will also draw the plot of the throughput but in KB/sec.





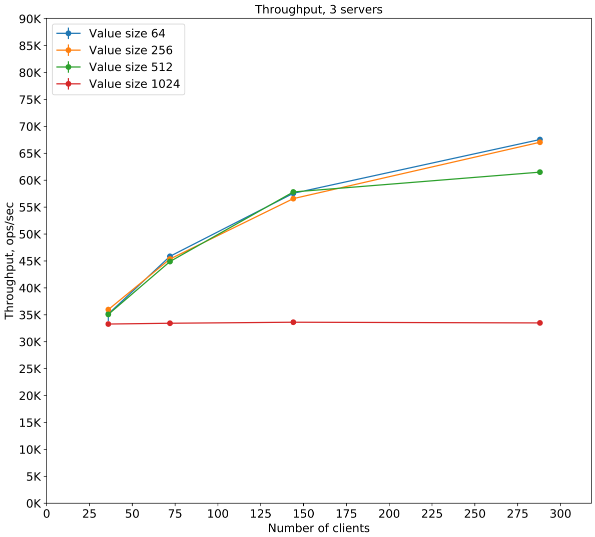
We see that red and green lines reach almost similar throughput which is close to 11MB/sec - 12MB/sec. Then we look on the server bandwidth, input and output bandwidth are different: 12MB/sec output and 25MB/sec input, we are bounded by output bandwidth. This is exactly that border, which our clients approached. However we see that configurations with less value size does not reach the same border, but are saturated in the end (32 VC). We address here to the server CPU utilization. By increasing the number of clients, we see that utilization increases up to 96 percents with 16 clients. So here we reach the saturation in clients. This is what we see from the plot, where we change the number of clients.

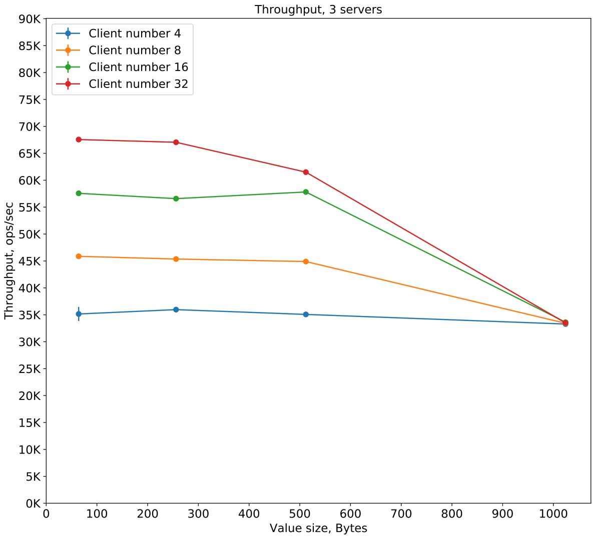
2.2 Three Servers (50 pts)

2.2.1 Setup (5 pts)

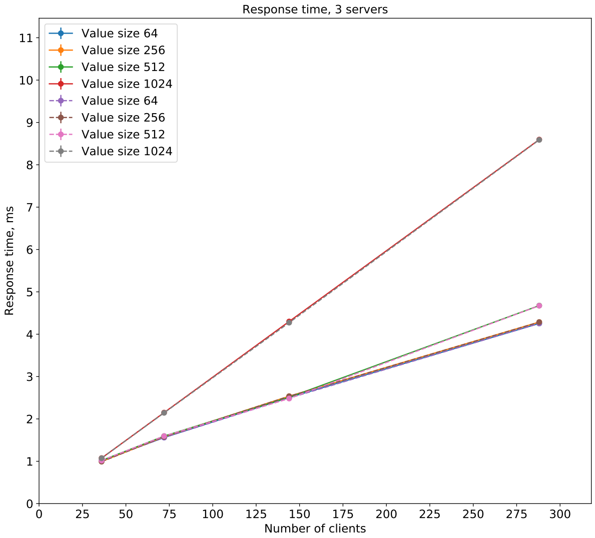
The setup was described in previous section. The plot trends and saturation points description is included either in Result analysis or in Explanation section.

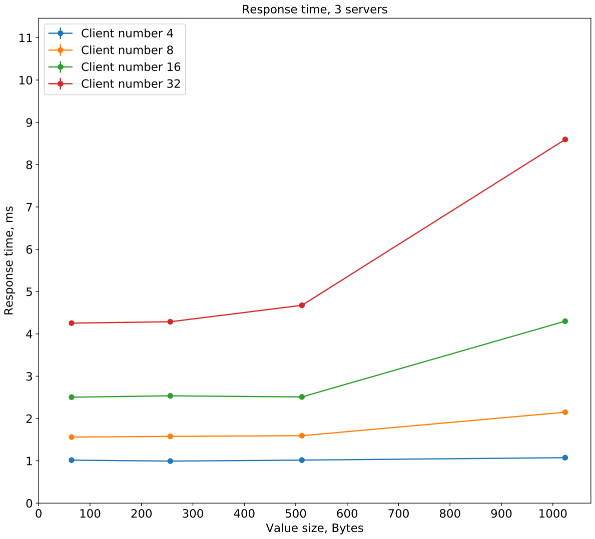
2.2.2 Throughput (10 pts)





2.2.3 Response Time (10 pts)





2.2.4 Result Analysis (5 pts)

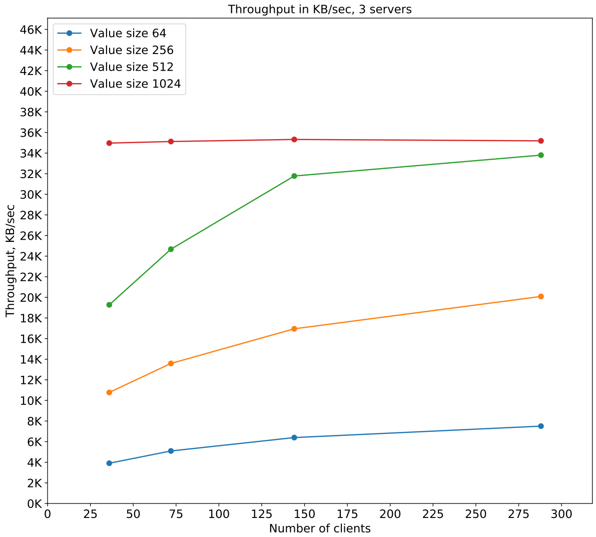
The results are explained with interactive law and the theoretic response number is also plotted on the response time graph.

We see here also the same situation with the response time and the same explanations as in section 2.1.4 are applicable. The only difference is that the initial throughput became higher due to the increase in number of servers, but the graph behavior stayed the same.

2.2.5 Explanation (20 pts)

For the 1024 value size graph we see, that the system is always in a saturated phase. When the value size is equal to 512, we see the undersaturated phase, but we could predict that the last point is already a point of saturation (we see the confirmation for this further when looking on the plot for throughput in KB/sec, where this plot almost reaches the network bandwidth). For the last two value sizes we see undersaturation, the throughput grows with the clients increasing.

The changes in response time correlate with the throughput. The throughput plot is not as steep, as it could be in a completely not saturated phase, so the response also does not stay constant and has a more dramatic growth for the completely saturated parts of the throughput and less dramatic for the partially saturated ones.



On the figure above 2.2.5 for the values sizes 512 and 1024 we see that they nearly approach 36KB/sec when increasing the number of clients. This is three times more than in previous section and this is what we expected to receive, when increasing three times the number of servers. And we are again bounded by the bandwidth (36 MB/sec in this case for 3 servers). This we see on the KB/sec plot. Also, for the plot in ops/sec we could see, that the throughput changed exactly in three times, because in the 1 server configuration with 512 and 1024 value size payload system was saturated.

For the other two values we see there that they approach saturation point (we don't clearly see it here) earlier than on 36KB/sec. According to additionally conducted measurements this is not due to the bandwidth limitations or cpu utilization (all clients are utilization for 50 percents, all server no more than 60 percents). We could argue, that this may be due to network latency influence, which prevents the number of requests grow higher. The same reason (some additional network latency) may also play a role for constant response time increase on the response time graphs.

The last mentioned fact is actually a reason why the increase of the value size (apart from 1024 value size) does not change anything regarding the throughput for all client numbers. The network latency prevents the throughput to be bigger for smaller value sizes, than for the bigger one and on the plot for throughput - value size and response time - value size we see steady regions.

2.3 Summary (20 pts)

	Maximum Throughput	Corresponding Response Time	Configuration (number of clients, msg. size)
One memcached server	37098.59 ops/sec	7.79 ms	64 Bytes, 32 VC
Three memcached servers	6755.63 ops/sec	4.25 ms	64 Bytes, 32 VC

2.3.1 Bottleneck Analysis (10 pts)

The bottleneck in these experiments is the output bandwidth of the server on the one side for large values sizes and the server's processing resources with high client numbers. The maximum throughput in ops/sec is not the same for all configurations, as we increased the number of servers and now requests are processed faster. However, for 1024 value we cannot go beyond the 12 MB/sec and 36 MB/sec border in 1 server and 3 servers cases, thus we have limitations also in ops/sec.

Also one of the bottlenecks is an additional network latency which prevents for low value size numbers having a high throughput in ops/sec even for small number of clients.

2.3.2 One and Three Servers Configurations (5 pts)

Three servers increases the throughput in Bytes/sec in three time for the configurations with 512 and 1024 Bytes. This happens on the one side due to the high payload size, which allows us to reach saturation, on the other side due to the network limitations. For 512 value size we need more clients in order to approach the network bandwidth, then in case of 1024 value size. On the opposite, when the number of client is small, we see that the throughput is not increased significantly, 1 and 3 servers configuration, as the system has a low load and even one server was enough to cope with the all the requests. We also introduced arguments about latency in the network during the analysis.

3 Baseline with Middleware (240 pts)

3.1 One Middleware, One Server (50 pts)

3.1.1 Setup (5pts)

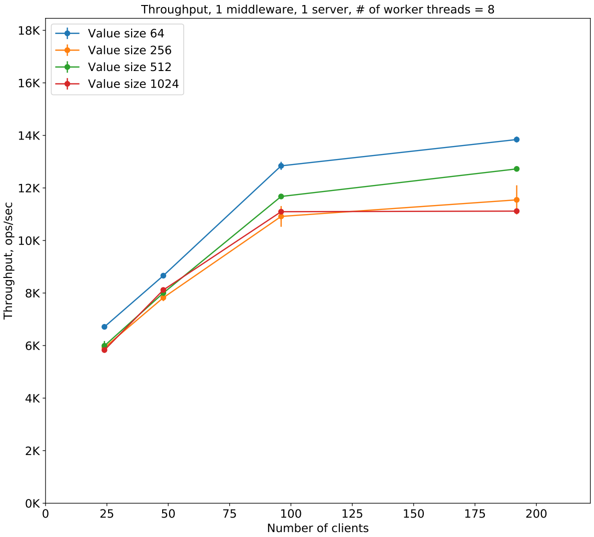
Here will be description of the setup for all following sections with middleware.

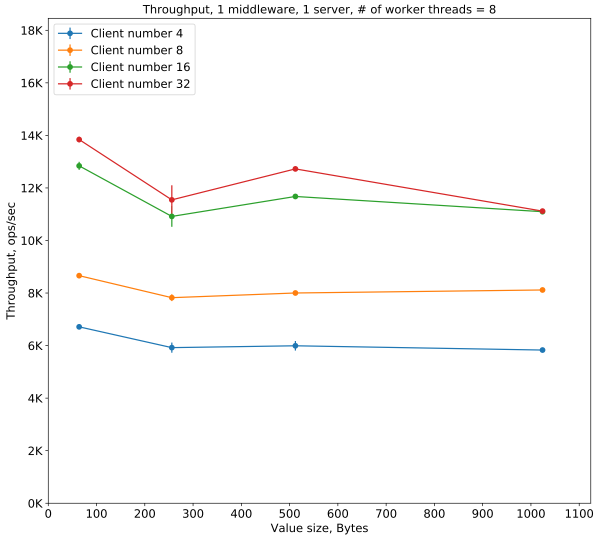
The same deployment was used in all experiments in the report. There was a global loop over value sizes, for each value size of global loop was run a nested loop of depth 3: firstly over worker threads, secondly over client number and inside over 3 repetitions. In each loop iteration the first client makes six connections (to the other two clients and to itself, for each client two connections) and launches script experiment.sh, passing there necessary parameters.

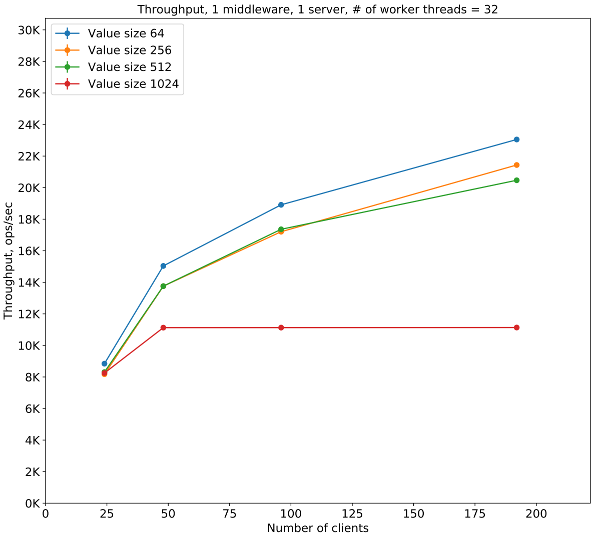
In experiment.sh script one memtier instance is launched. Each memtier instance on one client connects either to the same (deployment with one middleware) or to different middlewares (one memcached instance per Middleware). Then the first client waits until processes on all machines finish, waits additionally 10s and then runs next experiment iteration.

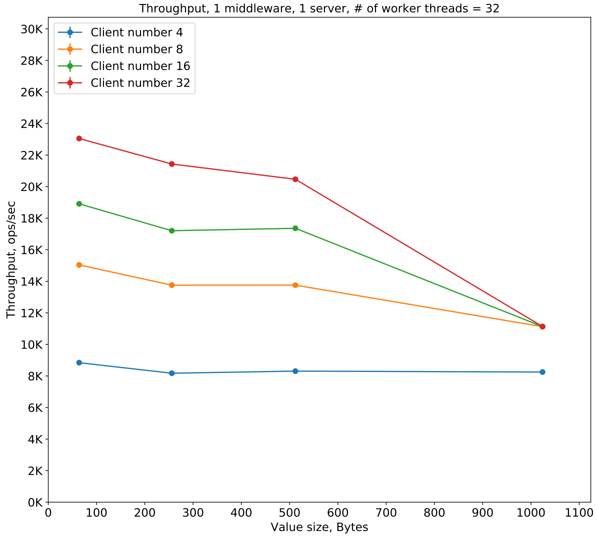
Also for each change of worker threads, first client restarts middleware(s) and waits 10 seconds to be sure that the restart completed. Each middleware accepts as a parameter one server or three servers.

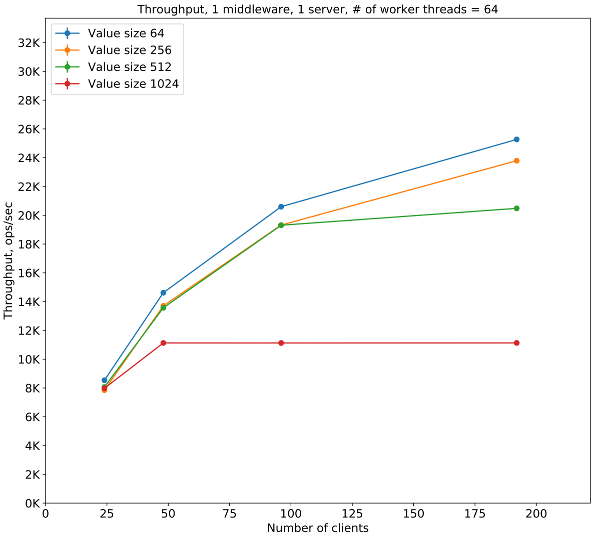
3.1.2 Throughput (10pts)

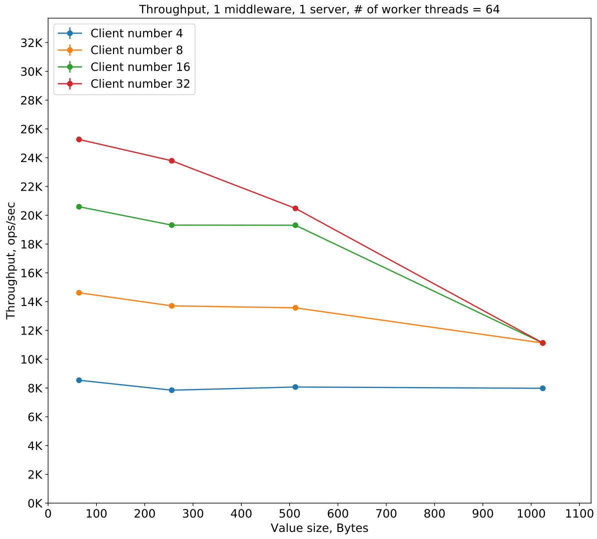

Throughput, 1 middleware, 1 server, # of worker threads = 8


Throughput, 1 middleware, 1 server, # of worker threads = 8

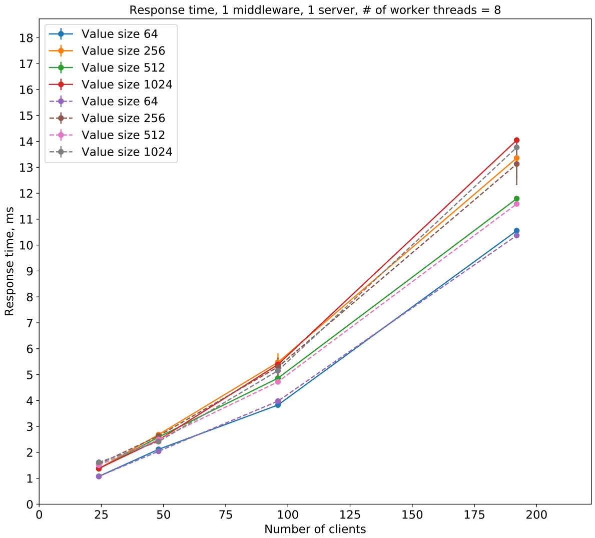

Throughput, 1 middleware, 1 server, # of worker threads = 32

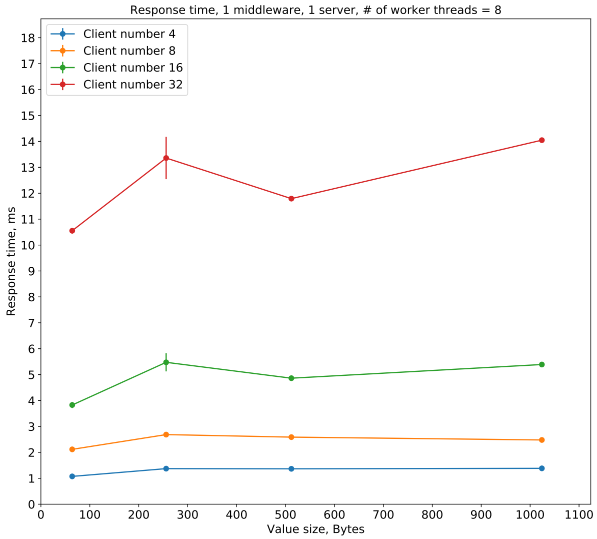

Throughput, 1 middleware, 1 server, # of worker threads = 32

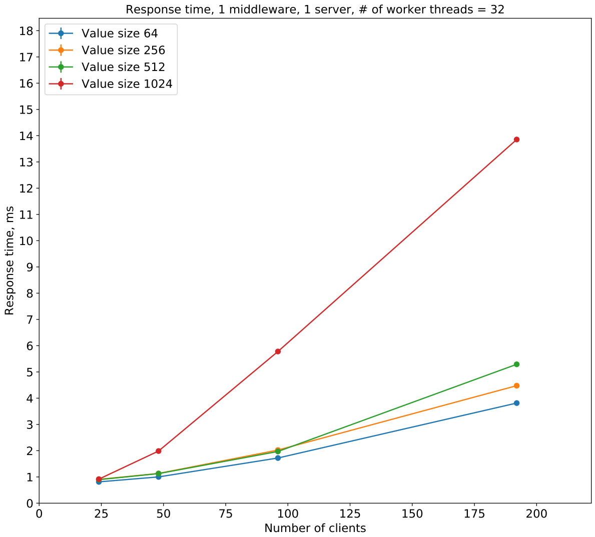

Throughput, 1 middleware, 1 server, # of worker threads = 64

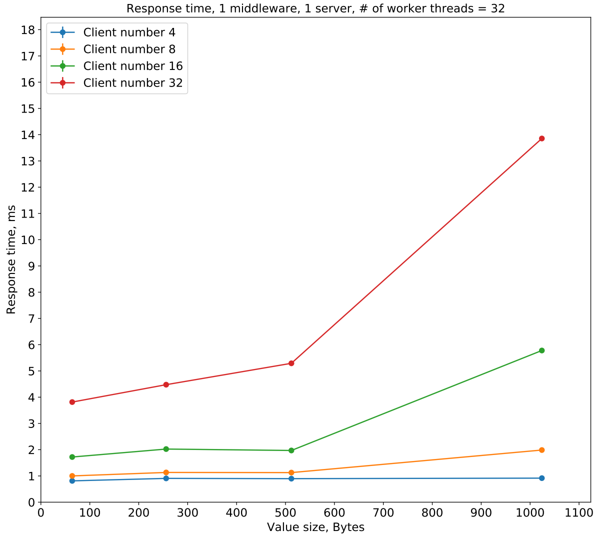

Throughput, 1 middleware, 1 server, # of worker threads = 64

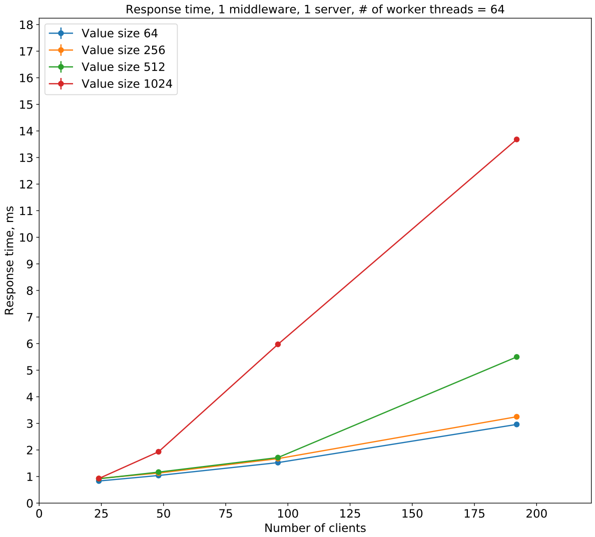
3.1.3 Response Time (10pts)

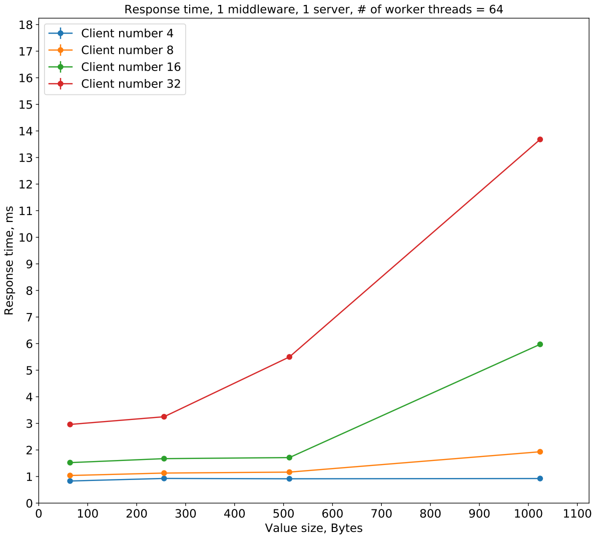

Response time, 1 middleware, 1 server, # of worker threads = 8


Response time, 1 middleware, 1 server, # of worker threads = 8


Response time, 1 middleware, 1 server, # of worker threads = 32

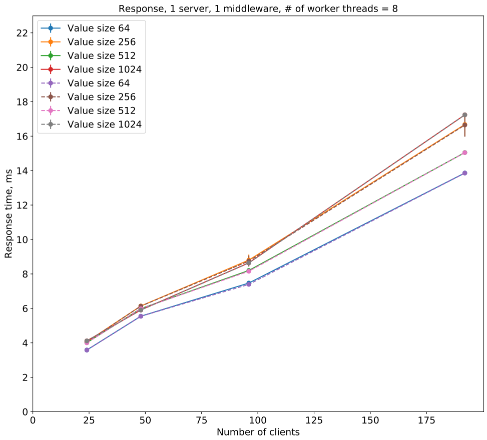

Response time, 1 middleware, 1 server, # of worker threads = 32


Response time, 1 middleware, 1 server, # of worker threads = 64


Response time, 1 middleware, 1 server, # of worker threads = 64

3.1.4 Result Analysis (5pts

The results are explained by the interactive law. However here from the middleware prospective we should add a rather high thinking time - it changes depending on the number of clients. According to the additional experiments, it should be around 2.5 ms for 4 virtual clients and varies around 3.5 ms for the rest number of clients. There is no completely precise way to measure this think time, as with ping or qperf it is hard to model the same client configuration, so it may give lower or higher numbers. So, for the middleware plots the interactive law will be not precise. We show additionally he proof for this law on the client side. Currently there is a proof with on the one plot for the response time (dotted lines).



We could measure the time on the memtier, which takes for request to come to middleware and for the empty answer sent from there. This time may be overestimated, but it takes actually into account those time periods that the response time on the middleware is not able to include (as we don't measure the time of the request processing before we actually receive it with take() function and the time that it still spend on the middleware after send() function).

So we will follow the mean numbers for one worker configuration as generally these client-middleware latency changes with varying the number of worker threads. For other configurations we could verify the interactive law with the help of the client data.

Interactive law is verified once for all sections with middleware, theoretical graph is shown on the response plot.

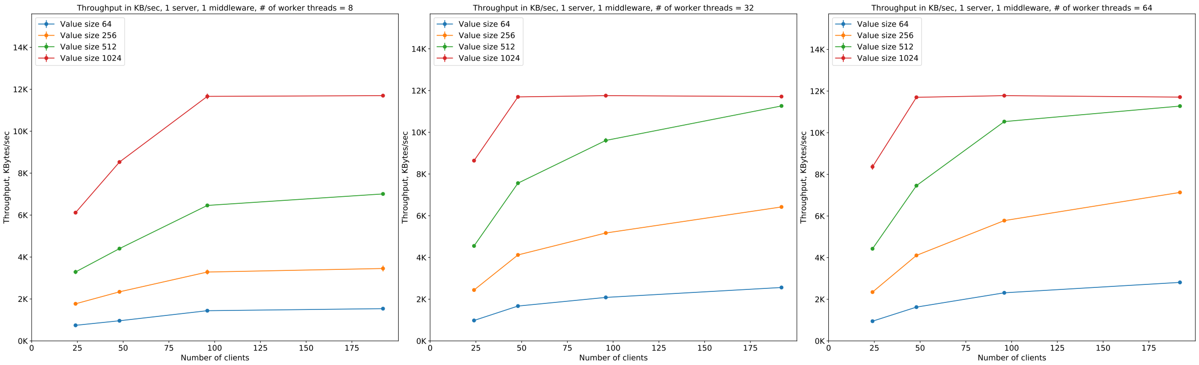
Generally here and in all other configuration as the response generally grows (as well as in section without middleware) and te throughput increase even from the very beginning is not equal 2x with the client number increase in 2x, we will take into account plots Response time - Throughput, which were shown one time in Section without middleware 2.1.5 and were also built for each configuration with middleware. These plots according to observations show the saturation point normally in the same place as the throughput shows.

3.1.5 Explanation (20pts)

In plots with 32 worker threads we see system with all value sizes apart from 1024, undersaturated (for 1024 value size the saturation come at 8 VC). But for the 512 Bytes plot we could actually see that the last point(32 VC) is already a point of saturation. We can confirm this by looking on the next plot for 64 worker threads,where for 512 Bytes the system reaches the same throughput faster with the increase of clients (16 VC) and that for 32 VC the value is the same. For 64 worker threads the system reaches a saturated state for both 512 (8 VC) and 1024 bytes (16 VC). For other two worker value sizes the system is always in undersaturated phase.

This corresponds to the response time behavior. For 8 worker threads, where we don't receive a high throughput growth with the increasing number of clients, we see the a rather steep growth for response time values with the growth of client numbers, comparing to other worker threads graphs. For 1024 value size due to very early saturation the response time starts to grow dramatically from 8 VC.

For further explanations of this behavior, we will draw plots for throughput in KB/sec (these plots are received using values from memtier)

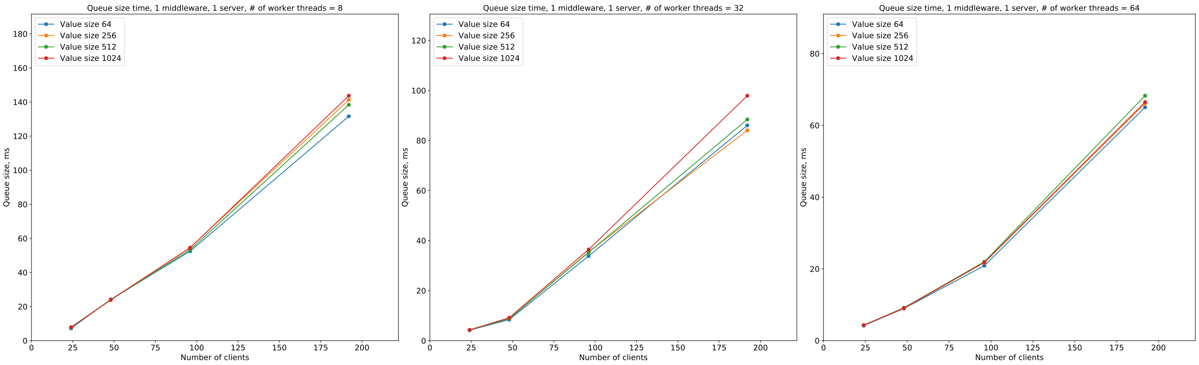


In all configurations the system with value size 1024 is operating on the maximum possible speed in KB/sec, as we are bounded again by the network. On the throughput plots in ops/sec we see that for 1024 value size the maximum throughput is also always the same, we reach saturation there. We also could see that the green plot for 512 bytes also almost reaches the system bandwidth limitation, thus we also see it's complete saturation in throughput plots for 32 (one last point) and 64 workers.

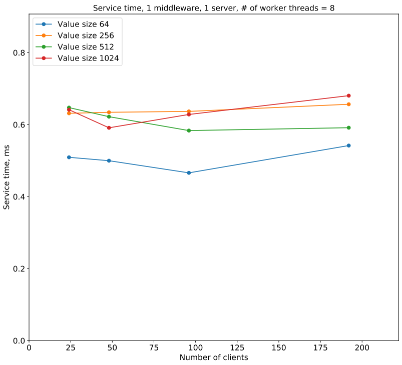
We could also compare these plots with the plots in Section 2.1.2 and see that the throughput numbers are the same for 1024 value size. This means, both with middleware or without for this value size it doesn't take a long time for system to saturate (without middleware it is already saturated, with it takes some time, as middleware introduces additional overhead).

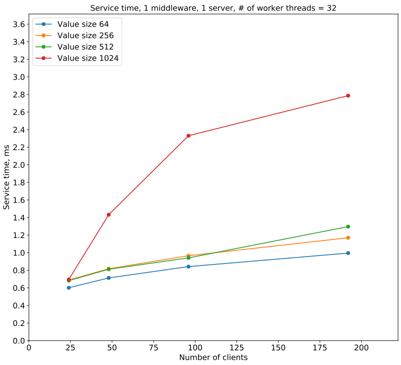
For 8 workers, we see the saturation for all value sizes starting from 16 VC, due to some errors in measurement it may be that plots for 512 and 256 Bytes lie close to each other. This confirms both plots for response time, where there is a significant growth with the client number increase from the very beginning and plot in KB/sec 3.1.5. This is due to the fact that middleware workers introduce additional overhead and could not cope with large amount of data fast enough. We also the confirmation for this, when the throughput increases and we go away from the saturation with the increase of worker threads.

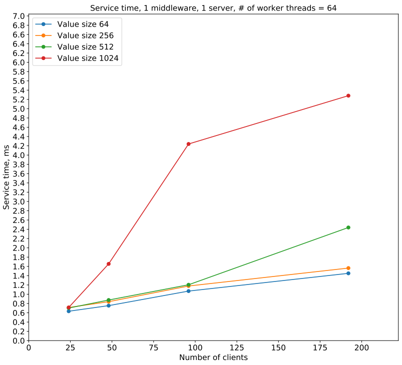
We could relate our analysis for 8 and 32, 64 worker threads difference with the queue length.



Indeed, with the worker threads number increases, the queue size decreases dramatically for the 8 - 32 worker number change, which helps us to gain a bigger throughput and on around 32 with 32 - 64 worker threads, meaning that system did not benefit a lot form this. If we look also on service time numbers we could see that service time for 64 workers increases comparing to the 32 workers, which means that the network latency or possibly increased load of server (which sometimes reaches 70 percents) is the problem here.







The bottleneck here are worker threads, as the system benefits significantly when the number of worker threads is increased.

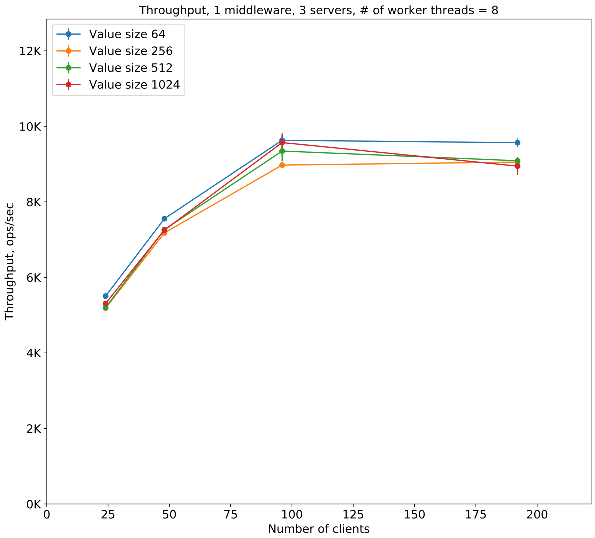
3.2 One Middleware, Three Server (50 pts)

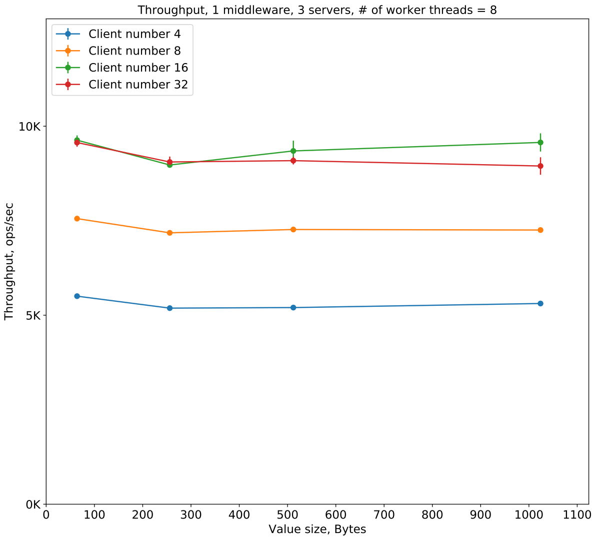
The saturation point description and plot description will follow in Explanation section.

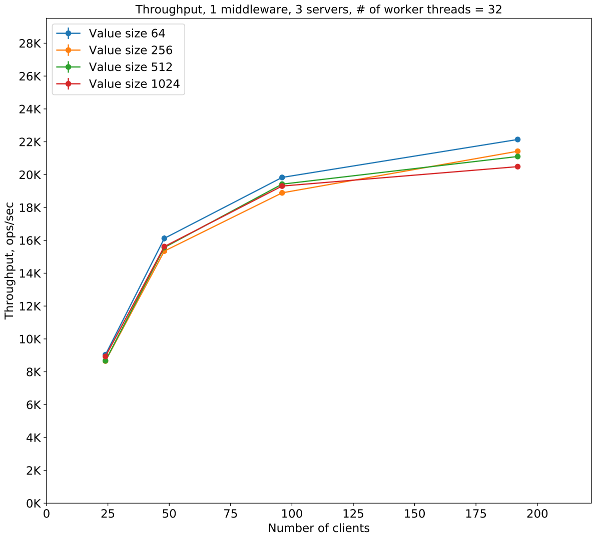
3.2.1 Setup (5 pts)

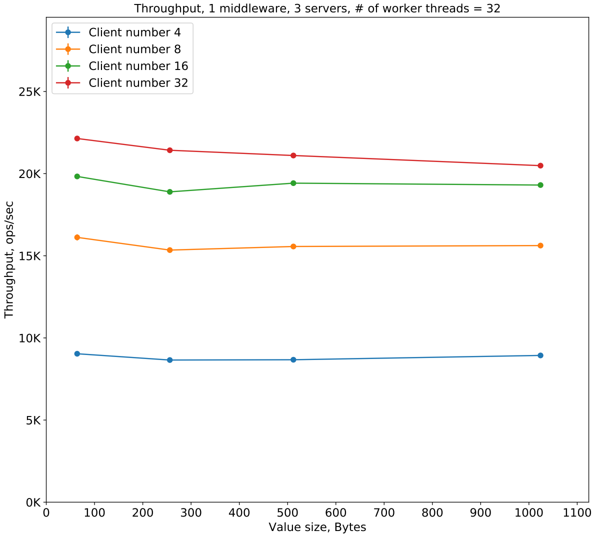
The setup was described in previous section.

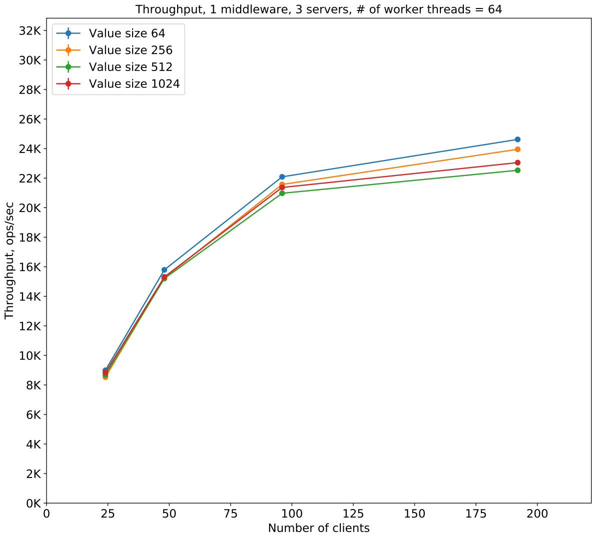
3.2.2 Throughput (10 pts)

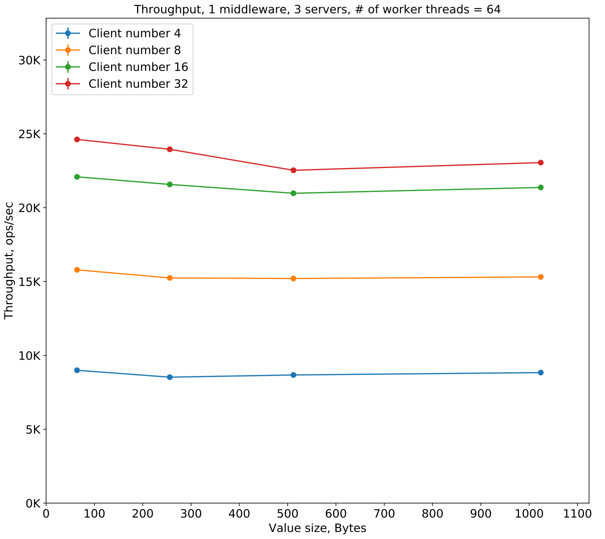




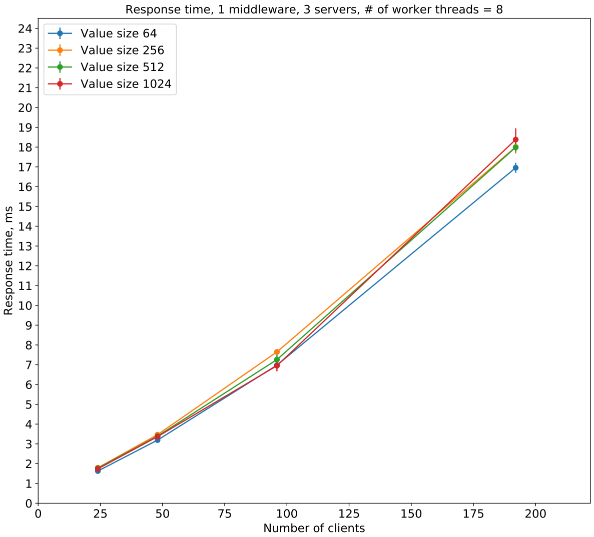


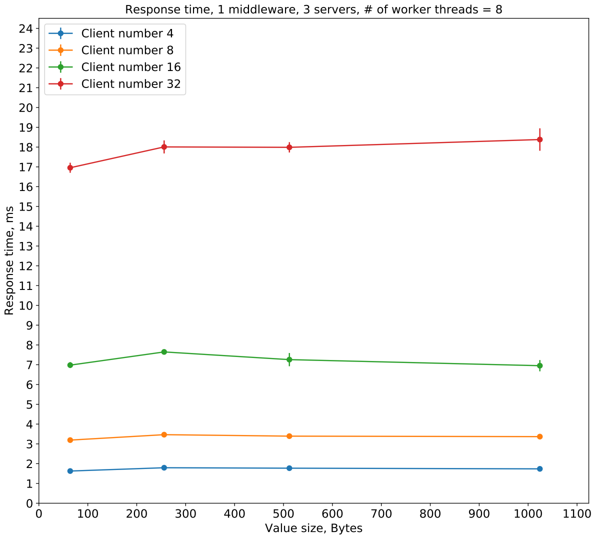


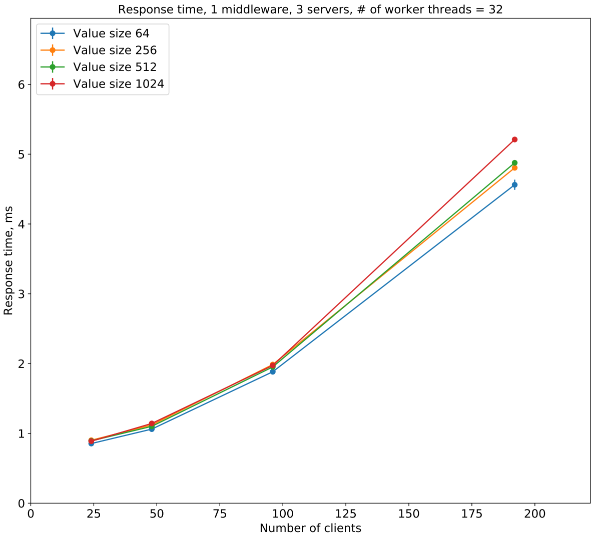


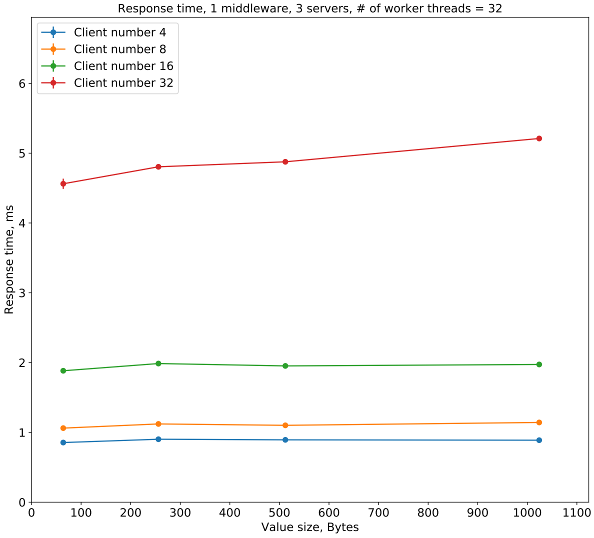


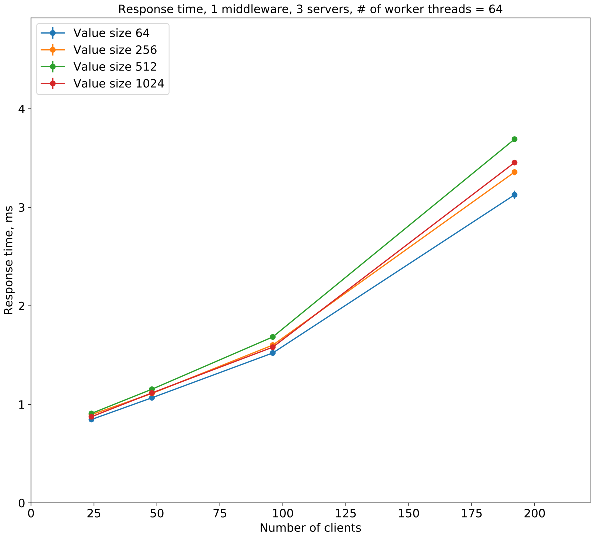
3.2.3 Response Time (10 pts)

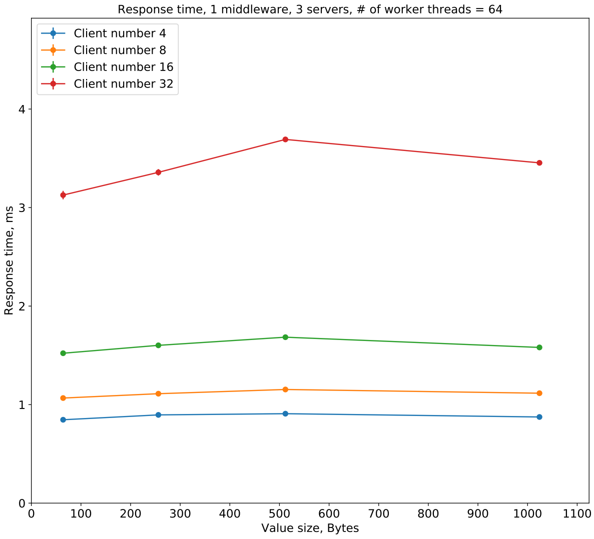

Response time, 1 middleware, 3 servers, # of worker threads = 8


Response time, 1 middleware, 3 servers, # of worker threads = 8


Response time, 1 middleware, 3 servers, # of worker threads = 32


Response time, 1 middleware, 3 servers, # of worker threads = 32


Response time, 1 middleware, 3 servers, # of worker threads = 64


Response time, 1 middleware, 3 servers, # of worker threads = 64

3.2.4 Result Analysis (5 pts)

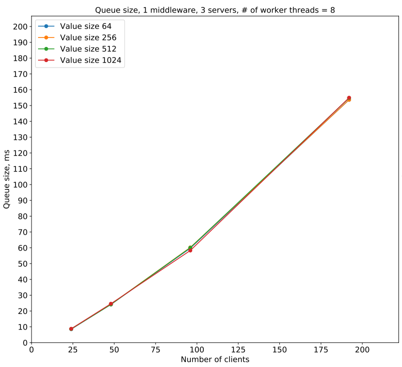
The interactive law for middleware was verified in previous section. The results didn't improve for the 32 and 64 worker threads and became a little bit worse from the throughput prospective for the 8 workers configuration. This correlates with the previous sections, the explanation will be presented in the next section.

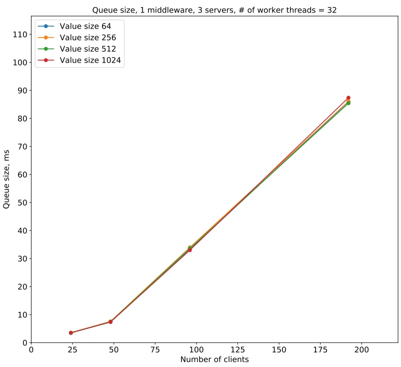
3.2.5 Explanation (pts 20)

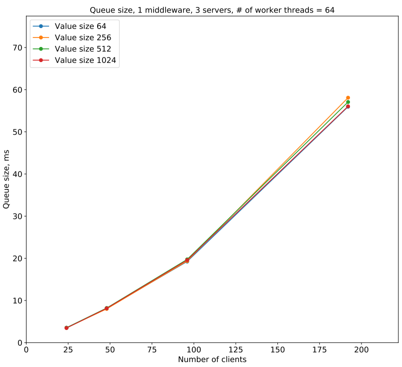
For 8 workers, system reaches saturation in 16 VC for all values sizes. For value sizes 1024 we see oversaturation point (32 VC). For all other worker threads numbers and value sizes we see system in an undesraturated state. However,32 VC may be the point of saturation for the system, as response already started growing dramatically.

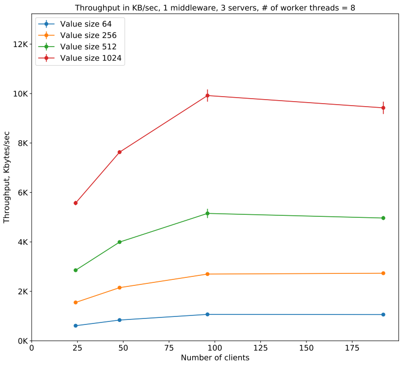
We also could observe, that with the growth value size throughput for the fixed client configuration stays constant. This means we are completely bounded by the

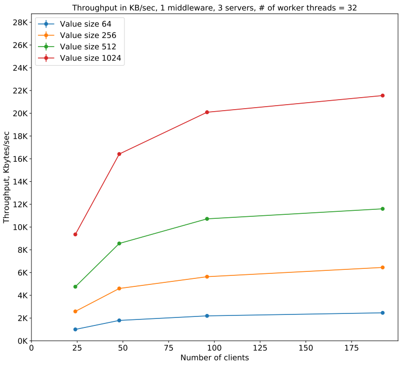
We see that the system benefits from the high number of worker threads, as the throughput increases more dramatically with the increase of clients comparing to the 8 worker threads. But there is no significant difference between 32 and 64 threads. With the increase of value sizes the throughput stays for each client configuration constant as the worker are able to hide due to parallelism the additional latency. The same applies to the response time. We could also illustrate this benefit on the queue length graphs:

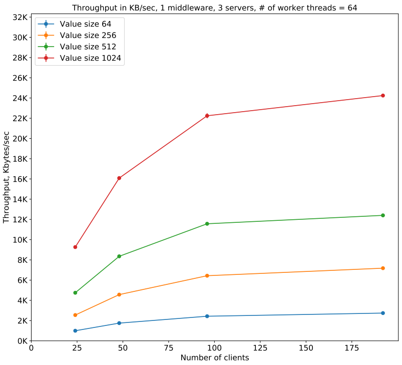




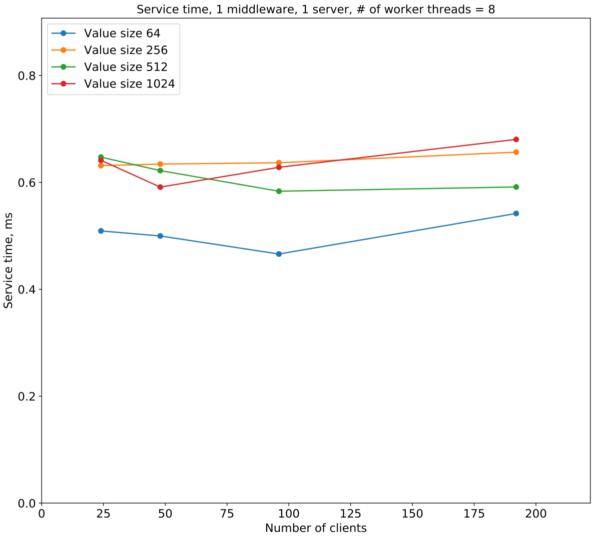


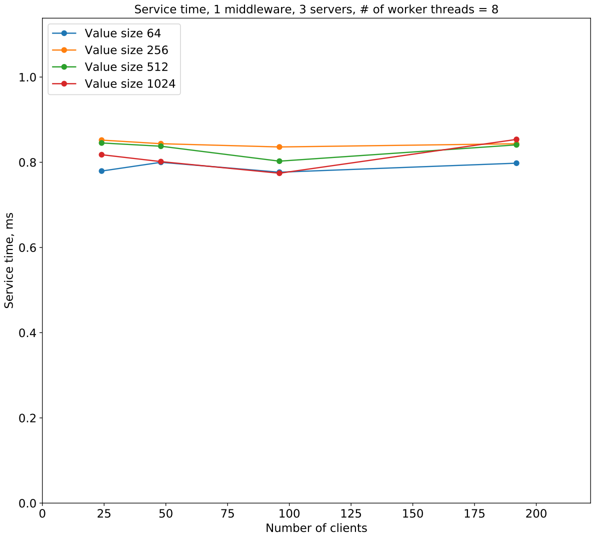






For the 8 worker threads we get decrease in performance comparing to the previous section. We could argue about it the following way: for the initial 8 worker configuration we observed all configurations in the saturation phases at the biggest number of clients or already approached to a saturation phase. As soon as this was not the maximum that the server could cope with and was totally bound by the worker threads, we actually should not expect three server to make the situation better as workers will operate on the same speed. But because of the fact, the servers' latencies are not equal and we, due to the latency analysis, added one server with a bigger latency than others, this resulted in bigger service time and as a result in worker throughput.





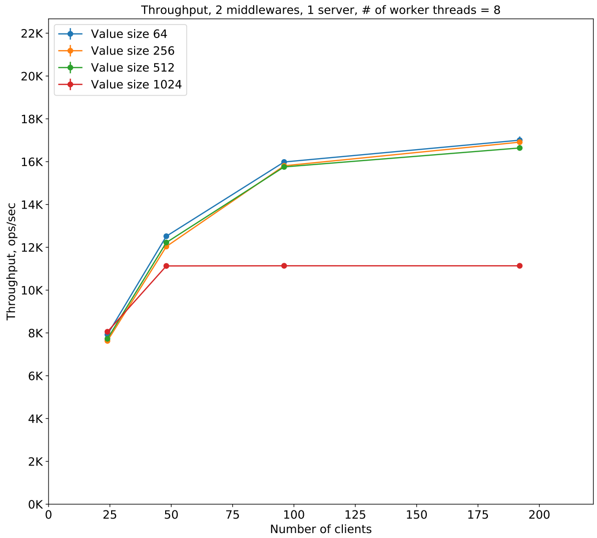
Changing value size does not impact on the throughput or response time, this means that worker threads are able to hide this latency due to parallelism. The same behavior we observed for low value size numbers in configuration 1 middleware - 1 server (for the bigger value sizes the system reached saturation and the workers threads operated at a possible maximum).

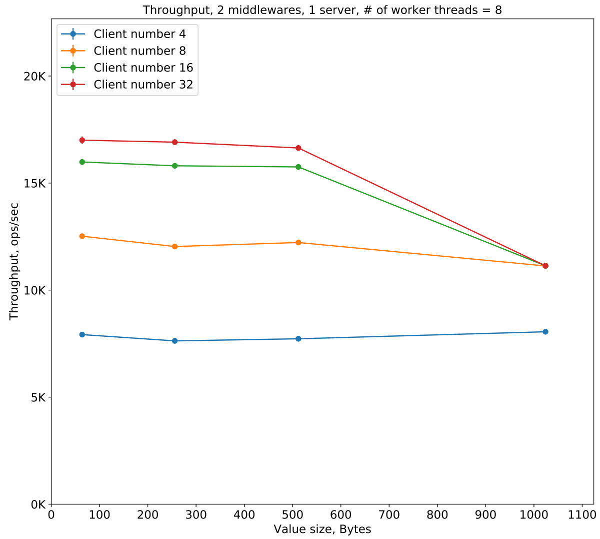
3.3 Two Middlewares, One Server (50 pts)

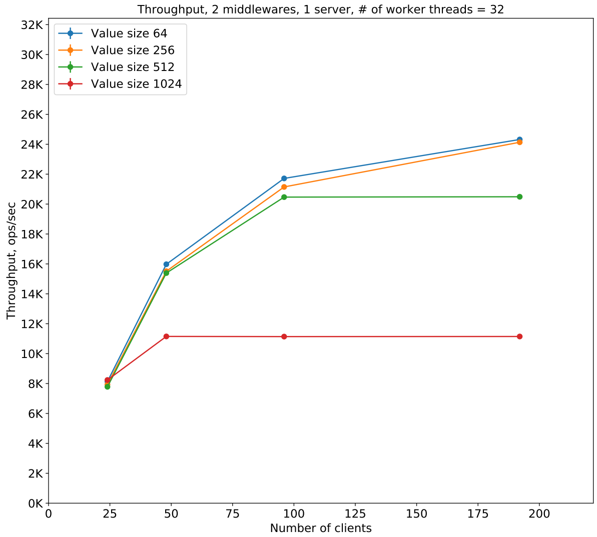
3.3.1 Setup (5 pts)

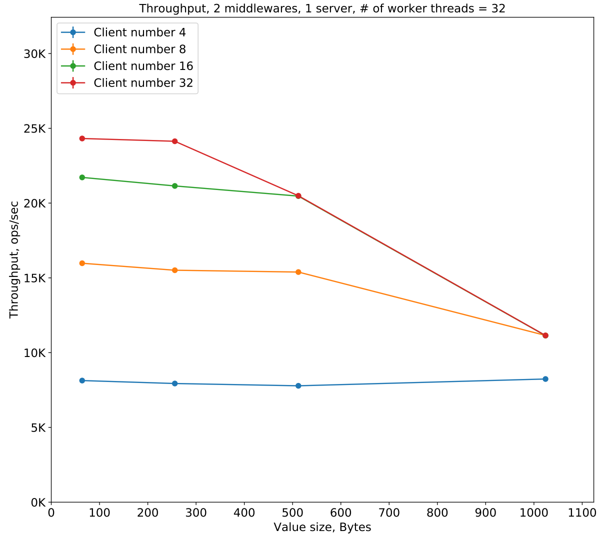
The setup was described in Section 3.1.1

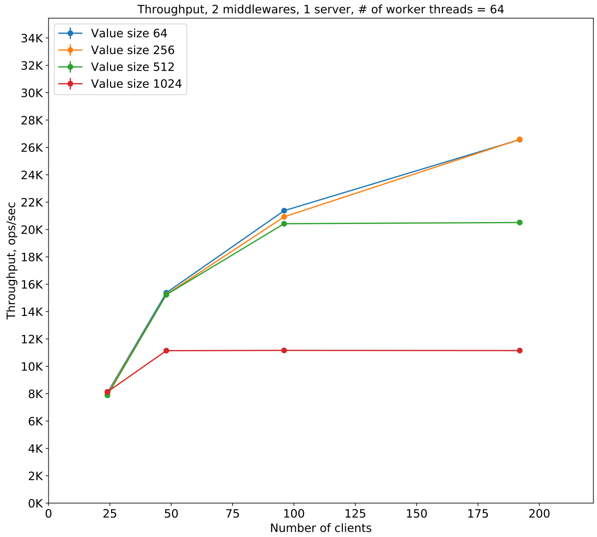
3.3.2 Throughput (10 pts)

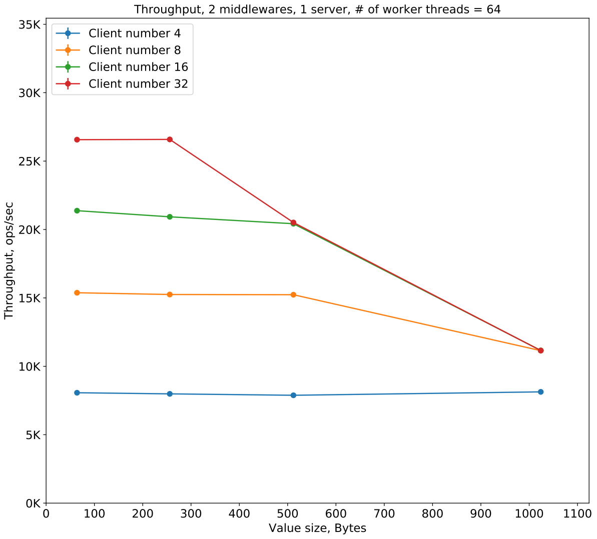

Throughput, 2 middlewares, 1 server, # of worker threads = 8


Throughput, 2 middlewares, 1 server, # of worker threads = 8

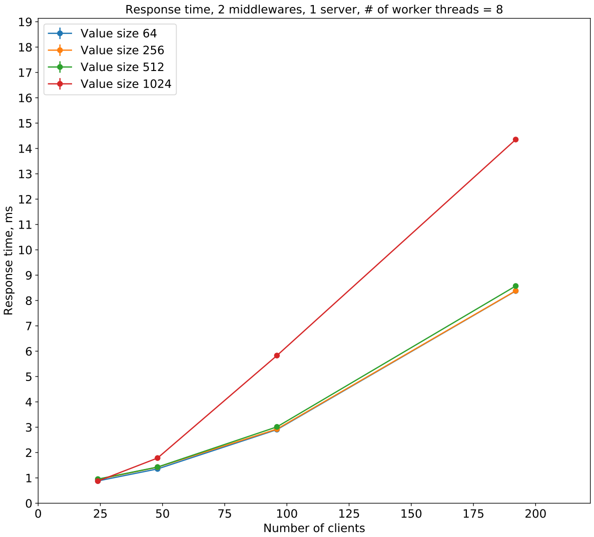

Throughput, 2 middlewares, 1 server, # of worker threads = 32

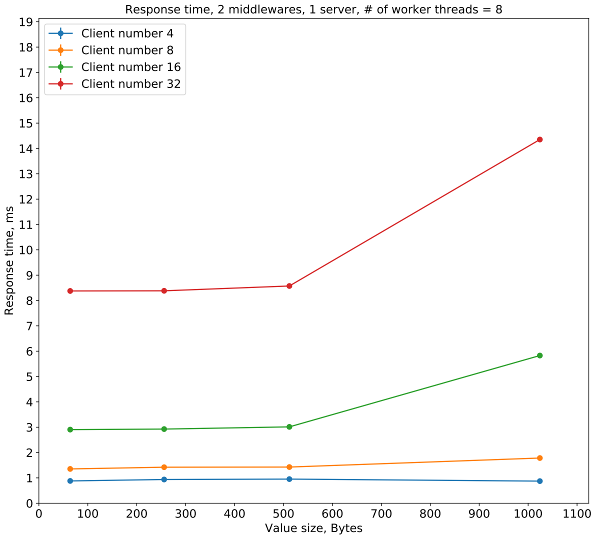

Throughput, 2 middlewares, 1 server, # of worker threads = 32

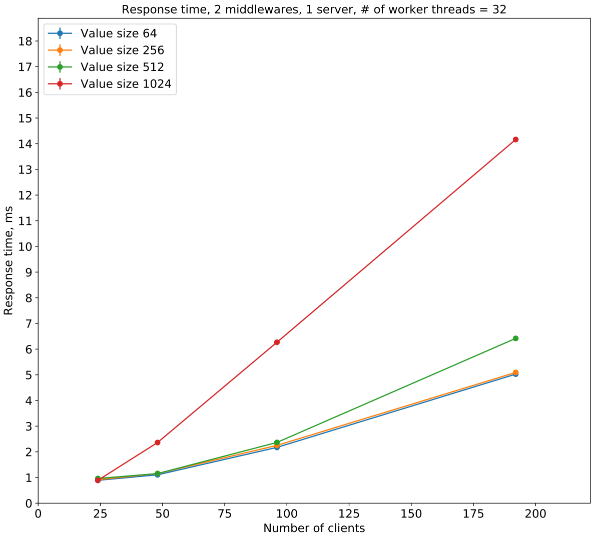

Throughput, 2 middlewares, 1 server, # of worker threads = 64

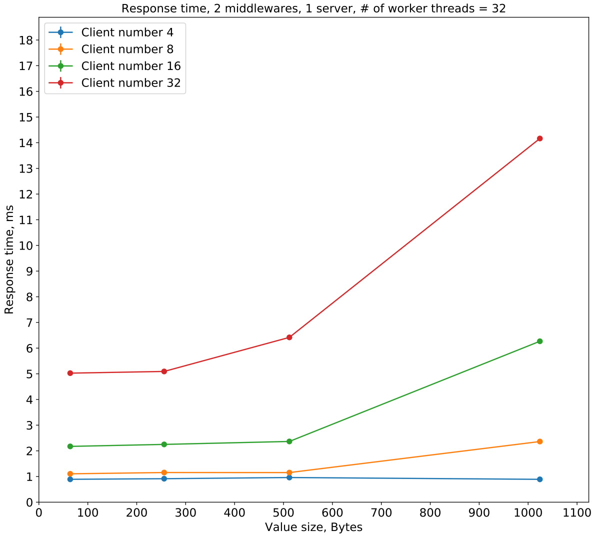

Throughput, 2 middlewares, 1 server, # of worker threads = 64

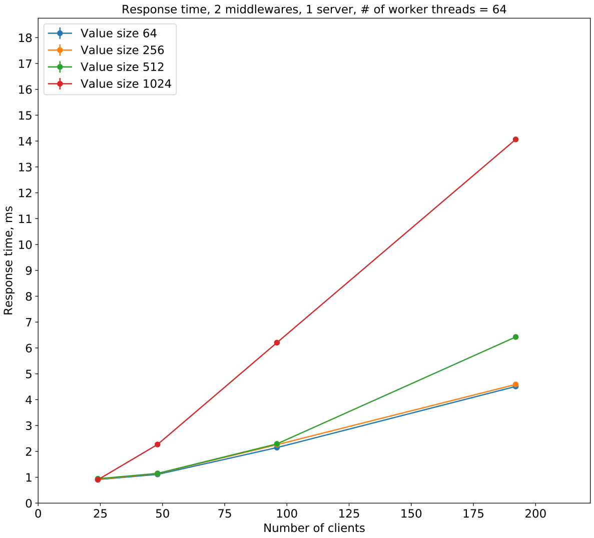
3.3.3 Response Time (10 pts)

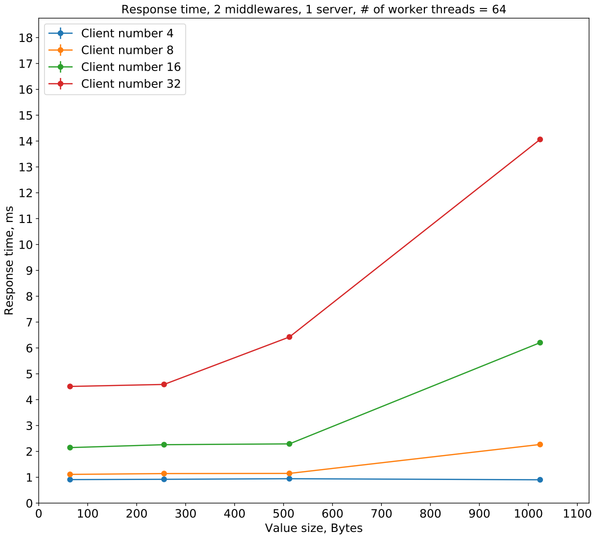












3.3.4 Result Analysis (5 pts)

Explain the results are explained by the interactive law (and why) and if they are consistent with previous experiments.

The results are consistent with the previously received ones, the raise of the throughput comparing to one middleware - one server configuration was expected, as we added one more level of parallelism.

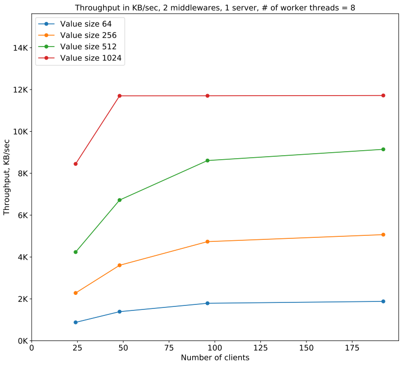
3.3.5 Explanation (20 pts)

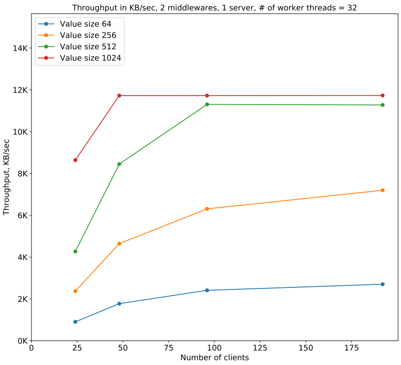
Due to more parallelism with the payload of 1024 Bytes the system reaches saturation with all worker threads rather fast - 8 VC. For 512 bytes payload the system is saturated for the worker threads 32 and 64, for the configuration with 8 workers for the first value sizes the system is undersaturated (reaching possibly the saturation point on the last value, as the response time and queue length start to grow dramatically)

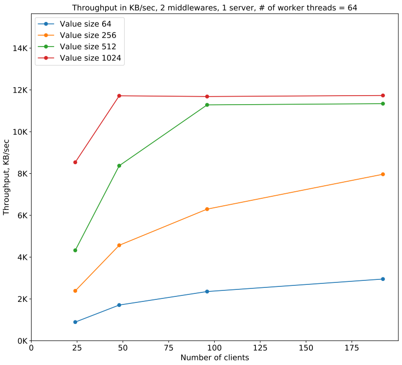
These observations correlate with response time plots. For 512 and 1024 value sizes the response time starts to grow faster after saturation point. For other two value sizes the growth of response time is higher for 8 workers configuration, as there we are almost approaching the saturation point and for other two worker thread numbers, where we are far from saturation, the response increases slowly and not significantly.

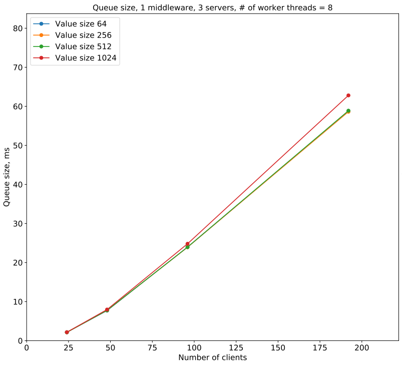
Increasing in value size, for low numbers of clients leaves the response and throughput in a steady state, this means that parallelism in system hides the additional latencies, introduced by the increased value size. For the values where this does not hold, we reach the saturation

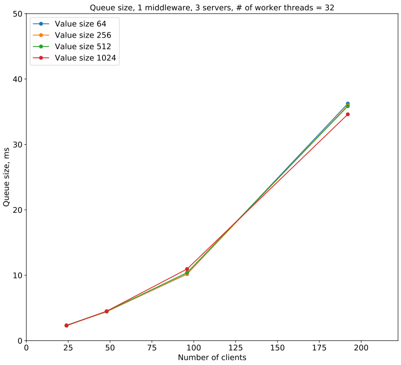
We could actually support here our suggestion in section with 1 middleware and three servers 3.2.2, that the value size 512 and 1024 reach their complete saturation (only network bounded) there on the last point as here the same value is reached in a clearly seen saturation phase.

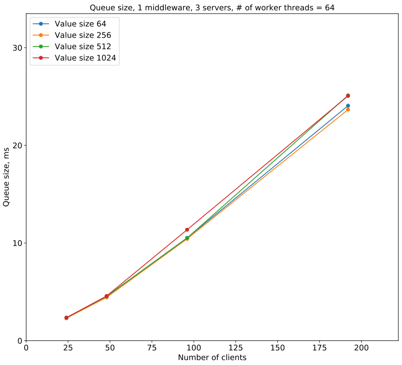












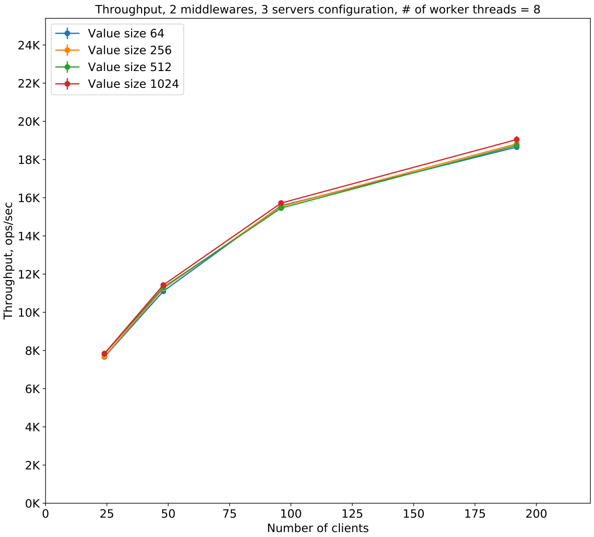
We see on the plots, that for 8 worker threads configuration even with 2 middleware the queue size increases rather fast which along with the fact that additional number of worker traverse the system from nearly saturated to a not saturated phase, confirms that worker threads are again a bottleneck, even though adding two middlewares increased the performance comparing to 1 middleware version.

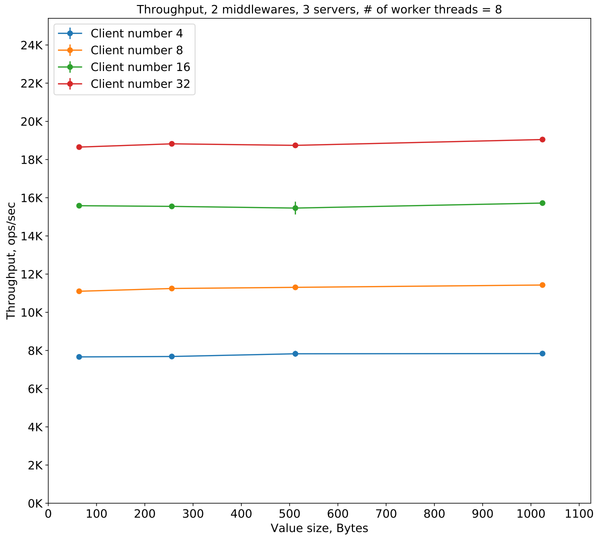
3.4 Two Middlewares, Three Server (50 pts)

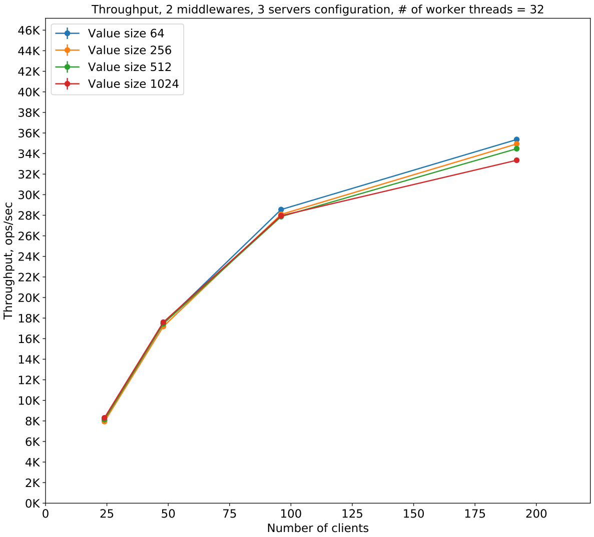
3.4.1 Setup (5 pts)

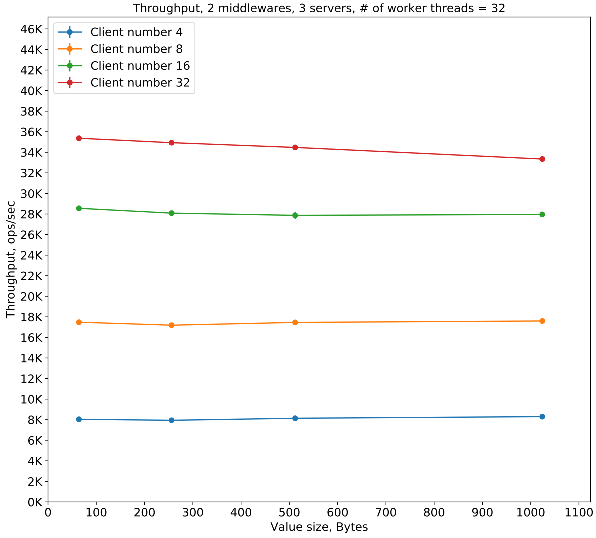
The setup was explained in 3.1.1.

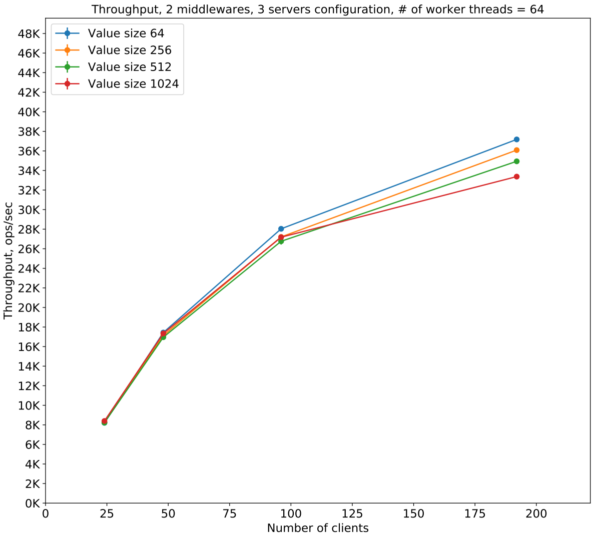
3.4.2 Throughput (10 pts)

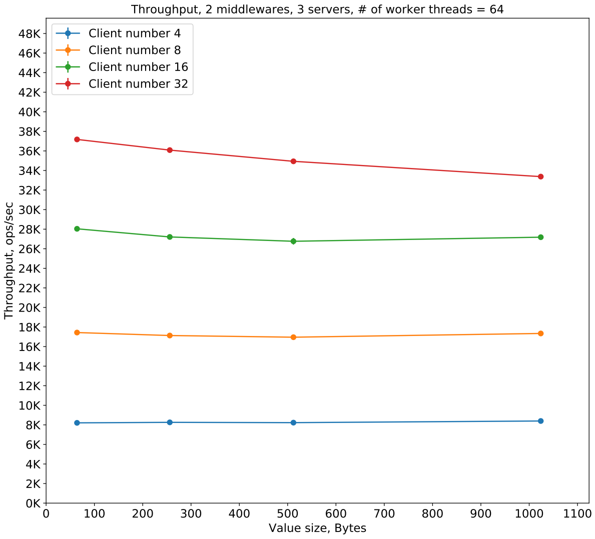

Throughput, 2 middlewares, 3 servers configuration, # of worker threads = 8


Throughput, 2 middlewares, 3 servers, # of worker threads = 8

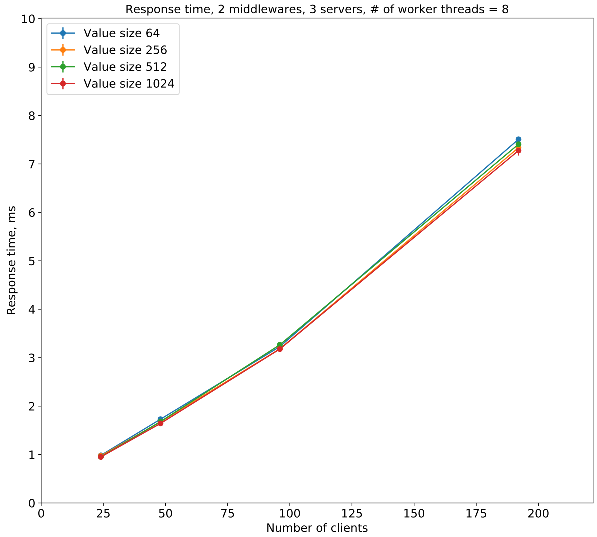

Throughput, 2 middlewares, 3 servers configuration, # of worker threads = 32

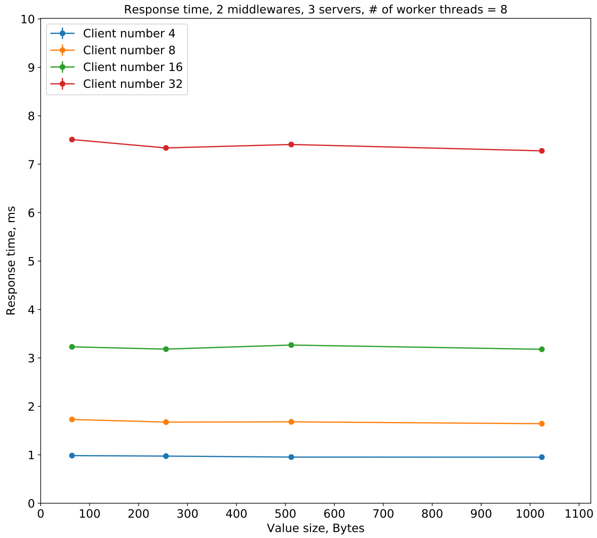

Throughput, 2 middlewares, 3 servers, # of worker threads = 32

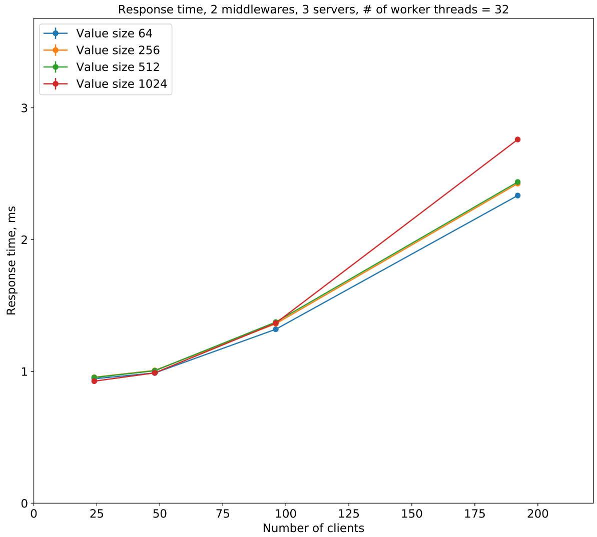

Throughput, 2 middlewares, 3 servers configuration, # of worker threads = 64

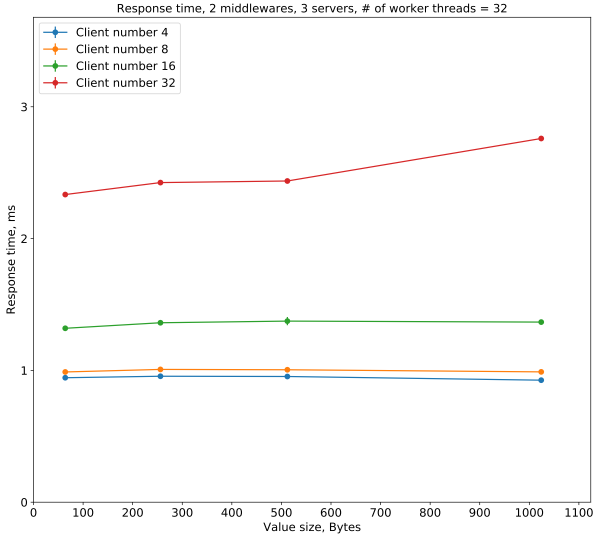

Throughput, 2 middlewares, 3 servers, # of worker threads = 64

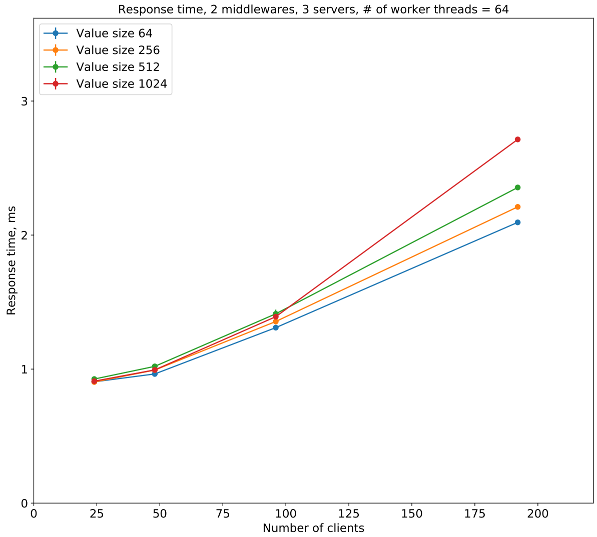
3.4.3 Response Time (10 pts)

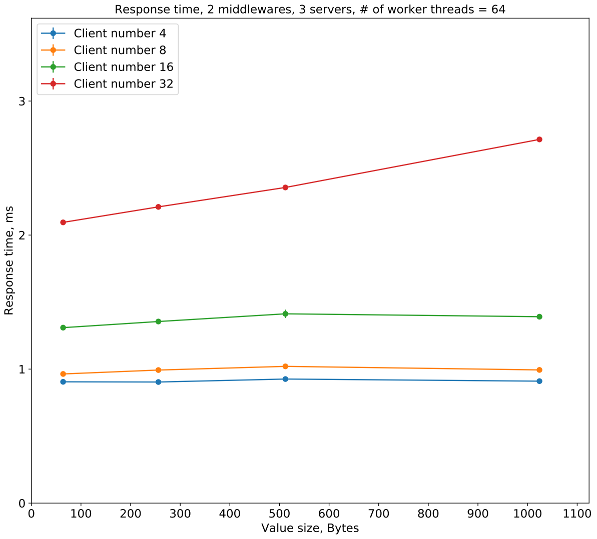










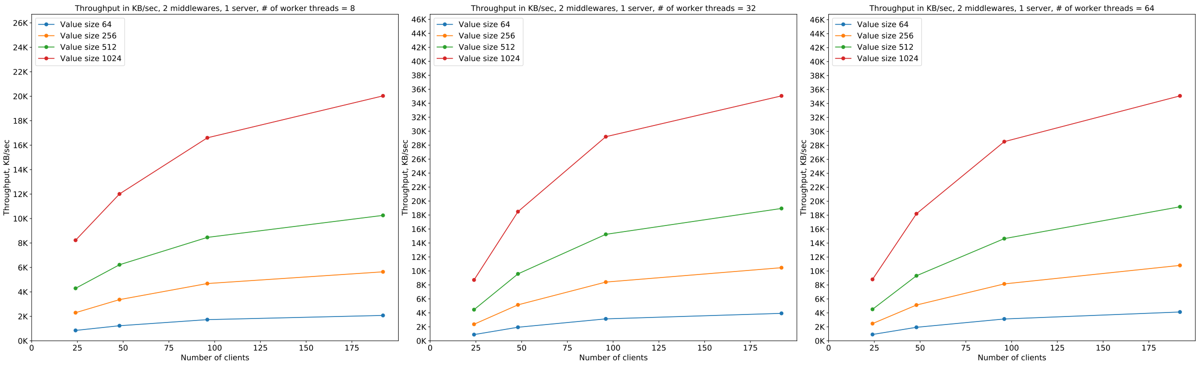


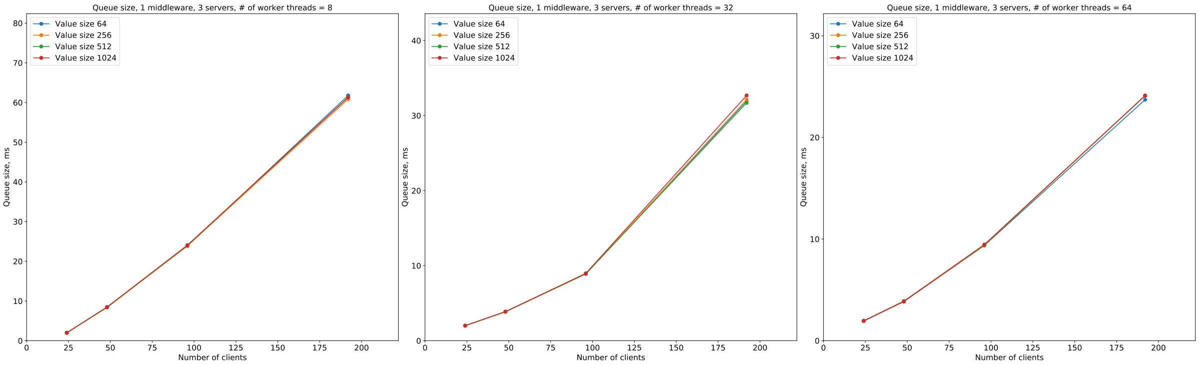
3.4.4 Result Analysis (5 pts)

The interactive law holds for the results.

The results correlate with the previous measurements. We see, that this configuration is the most beneficial. By using three servers we are not bound by 12MB/sec any more. Thus, by the combination of worker threads and middlewares the latencies from the increasing of value size disappeared completely (this confirms the plot for throughput and response time with changing number of value size on the x axis).

3.4.5 Explanation (20 pts)





Still due to the low number of worker threads the plot for 1024 value size doesn't manage to reach the saturation point of 36MB/sec (the network boundary for three servers).

With the transition to the 64 worker threads the service time increases dramatically for higher number of clients, especially for 1024 value size, as system in this configuration , is almost at a complete saturation determined by the network bound. For other value sizes the increase in value size may be explained by the heavy load of the servers in this configuration (up to 90 percents), so the server answer time increases.

3.5 Summary (40 pts)

Maximum throughput for one and two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Section 3.1: Measured on middleware				
Section 3.1: Measured on clients			n/a	
Section 3.2: Measured on middleware				
Section 3.2: Measured on clients			n/a	
Section 3.3: Measured on middleware				
Section 3.3: Measured on clients			n/a	
Section 3.4: Measured on middleware				
Section 3.4: Measured on clients			n/a	

3.5.1 Bottleneck Analysis (10 pts)

In different experiments we could observe different bottlenecks. For experiments with small number of worker threads we clearly saw on all graphs, that the worker threads are the bottleneck. When increasing the number of workers or adding the additional middleware during some experiments with the big value sizes, we could observer that the bottleneck is the network bandwidth, this is why we see saturation there. With high number of worker threads, we are able to observe the third factor which is limiting the performance - network latency or high server load.

So, to conclude when increasing the value size we generally either stay with the same throughput if the worker threads are able to hide additional latencies, we could see some drops in performance, if the value size is near reaching the saturation point and the service time increases and we could also a dramatic drop if for the higher value size values we transition to the saturation phase.

3.5.2 One and Two Middleware Configurations (10 pts)

In the one server configuration the system does not benefit signifantly from validating the secod middleware, as even though we manage to hide the latencies of the worker threads a little bit by doubling them,

3.5.3 One and Three Server Configurations (10 pts)

We clearly saw, that we don't benefit from adding the additional servers with worker threads number equal to 8, as the facts that now the throughput is dominated by the slowest server and the worker threads being bottlenecks themselves combine and we receive the smaller throughput.

For other configuration the system generally benefited from adding more servers, as the system was less bounded by the worker threads performance and could utilize more bandwidth.

4 2K Analysis (80 pts)

4.1 2K Model (20 pts)

In 2K model we compute the effect of 3 factors: number of servers, number of middlewares, number of worker threads. There are two possible models to use: additive and multiplicative. Two of our effects should be multiplicative, the number of workers and the number of middlewares. The number of servers is not multiplicative in relation to other two factors. Also, the ratio y_max/y_min of the values is not big enough, which is required for the multiplicative model.

Further we will see other arguments for applying the additive model. Even though the additive model seems to be more applicable here, we will investigate a multiplicative model as well, in order to show that in general it gives similar results.

Additive model The results we obtain here , will be also applicable in multiplicative model.

For these three variables we solve a linear equation where the effects are unknowns, the result is the measured throughput and the coefficients are determined the following way.

For every factor we introduce new variable, which is equal to 1 for greater factor value and equal to -1 for lower factor value.

Servers will be referred as factor A, middleware as factor B, workers as factor C. Further we see example of the newly introduced variable for factor A:

$$X_A = \begin{cases} -1, & \text{if 1 server} \\ 1, & \text{if 3 servers} \end{cases}$$

We can now write 2^k linear equations of the form (actually we apply here a linear regression model): $y = q_0 + q_A * x_A + q_B * x_B + q_C * x_C + q_AB * x_A * x_B + q_AC * x_A * x_C + q_BC * x_B * x_C + q_ABC * x_A * x_B * x_C$

The coefficients can be computed here easily by creating $8 * 8$ sign matrix:

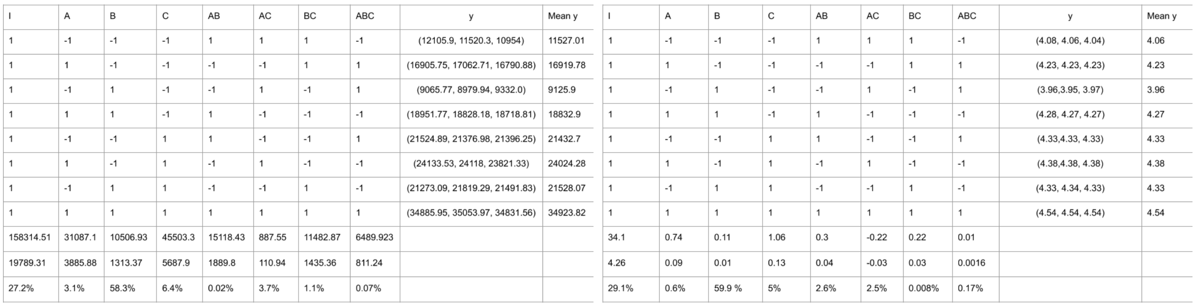
Then we can compute coefficients by multiplying the entries from the column I by the ones in column y, sum the received elements and put them under the column I. The same operation is done for all columns. The sums under each column are divided by 4. The result is the coefficient for every factor. Number under I is the mean performance.

In the end we are interested in percentage numbers, in order to decide whether the factor has significant effect on performance or not. Also we need to isolate the variation, which happens because of repetitions. In the end we could use the following formulas to compute:

$SSY = \sum_{i=1}^{2^k} \sum_{j=1}^{r} y_{ij}^2$, $SS0 = 2^k * r * q_0^2$, $SST = SSY - SS0$, $SSj = 2^k * r * q_j^2$, $SSE = SST - \sum_{i=1}^{2^k-1} SSj$, $(SSj/SST) * 100$.

These are sum of y squares, sum of squares of the mean y, total sum of squares, variation explained by factor j, variation explained by errors, percentage of variation of each factor accordingly.

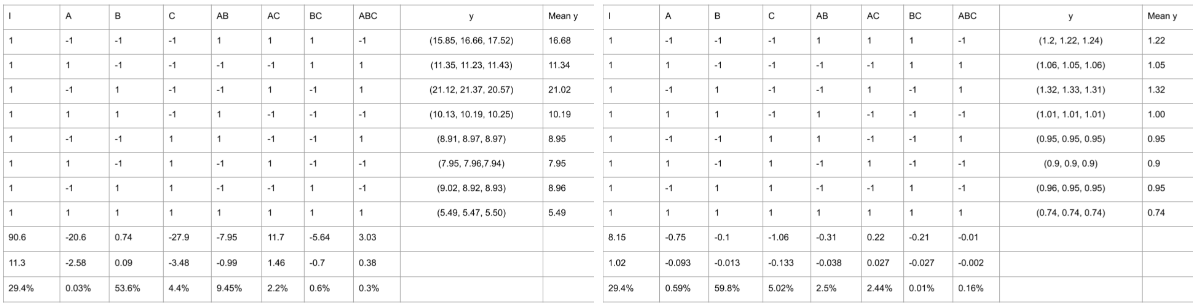
The following tables present experimental results. These results are obtained with the scripts, called Equation solver. There, we plug in the results of the experiments and the program computes the coefficients and percentage of variation for each factor.



I	A	B	C	AB	AC	BC	ABC	y	Mean y
1	-1	-1	-1	1	1	1	-1	(12105.9, 11520.3, 10954)	11527.01
1	1	-1	-1	-1	-1	1	1	(16905.75, 17062.71, 16790.88)	16919.78
1	-1	1	-1	-1	1	-1	1	(9065.77, 8979.94, 9332.0)	9125.9
1	1	1	-1	1	-1	-1	-1	(18951.77, 18828.18, 18718.81)	18832.9
1	-1	-1	1	1	-1	-1	1	(21524.89, 21376.98, 21396.25)	21432.7
1	1	-1	1	-1	1	-1	-1	(24133.53, 24118, 23821.33)	24024.28
1	-1	1	1	-1	-1	1	-1	(21273.09, 21819.29, 21491.83)	21528.07
1	1	1	1	1	1	1	1	(34885.95, 35053.97, 34831.56)	34923.82
158314.51	31087.1	10506.93	45503.3	15118.43	887.55	11482.87	6489.923		
19789.31	3885.88	1313.37	5687.9	1889.8	110.94	1435.36	811.24		
27.2%	3.1%	58.3%	6.4%	0.02%	3.7%	1.1%	0.07%		

I	A	B	C	AB	AC	BC	ABC	y	Mean y
1	-1	-1	-1	1	1	1	-1	(4.08, 4.06, 4.04)	4.06
1	1	-1	-1	-1	-1	1	1	(4.23, 4.23, 4.23)	4.23
1	-1	1	-1	-1	1	-1	1	(3.96,3.95, 3.97)	3.96
1	1	1	-1	1	-1	-1	-1	(4.28, 4.27, 4.27)	4.27
1	-1	-1	1	1	-1	-1	1	(4.33,4.33, 4.33)	4.33
1	1	-1	1	-1	1	-1	-1	(4.38,4.38, 4.38)	4.38
1	-1	1	1	-1	-1	1	-1	(4.33, 4.34, 4.33)	4.33
1	1	1	1	1	1	1	1	(4.54, 4.54, 4.54)	4.54
34.1	0.74	0.11	1.06	0.3	-0.22	0.22	0.01		
4.26	0.09	0.01	0.13	0.04	-0.03	0.03	0.0016		
29.1%	0.6%	59.9 %	5%	2.6%	2.5%	0.008%	0.17%		

Figure 1: Effects for throughput time, additive(right) and multiplicative(left) models



I	A	B	C	AB	AC	BC	ABC	y	Mean y
1	-1	-1	-1	1	1	1	-1	(15.85, 16.66, 17.52)	16.68
1	1	-1	-1	-1	-1	1	1	(11.35, 11.23, 11.43)	11.34
1	-1	1	-1	-1	1	-1	1	(21.12, 21.37, 20.57)	21.02
1	1	1	-1	1	-1	-1	-1	(10.13, 10.19, 10.25)	10.19
1	-1	-1	1	1	-1	-1	1	(8.91, 8.97, 8.97)	8.95
1	1	-1	1	-1	1	-1	-1	(7.95, 7.96,7.94)	7.95
1	-1	1	1	-1	-1	1	-1	(9.02, 8.92, 8.93)	8.96
1	1	1	1	1	1	1	1	(5.49, 5.47, 5.50)	5.49
90.6	-20.6	0.74	-27.9	-7.95	11.7	-5.64	3.03		
11.3	-2.58	0.09	-3.48	-0.99	1.46	-0.7	0.38		
29.4%	0.03%	53.6%	4.4%	9.45%	2.2%	0.6%	0.3%		

I	A	B	C	AB	AC	BC	ABC	y	Mean y
1	-1	-1	-1	1	1	1	-1	(1.2, 1.22, 1.24)	1.22
1	1	-1	-1	-1	-1	1	1	(1.06, 1.05, 1.06)	1.05
1	-1	1	-1	-1	1	-1	1	(1.32, 1.33, 1.31)	1.32
1	1	1	-1	1	-1	-1	-1	(1.01, 1.01, 1.01)	1.00
1	-1	-1	1	1	-1	-1	1	(0.95, 0.95, 0.95)	0.95
1	1	-1	1	-1	1	-1	-1	(0.9, 0.9, 0.9)	0.9
1	-1	1	1	-1	-1	1	-1	(0.96, 0.95, 0.95)	0.95
1	1	1	1	1	1	1	1	(0.74, 0.74, 0.74)	0.74
8.15	-0.75	-0.1	-1.06	-0.31	0.22	-0.21	-0.01		
1.02	-0.093	-0.013	-0.133	-0.038	0.027	-0.027	-0.002		
29.4%	0.59%	59.8%	5.02%	2.5%	2.44%	0.01%	0.16%		

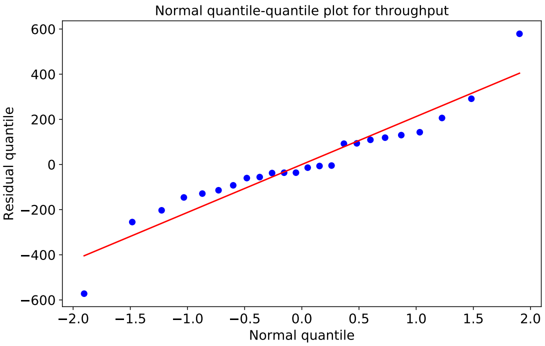
Figure 2: Effects for response time, additive(right) and multiplicative(left) models

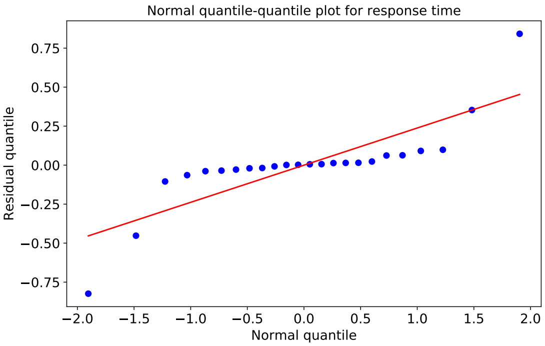
Validation of model applicability Applicability we can verify with following visual tests:

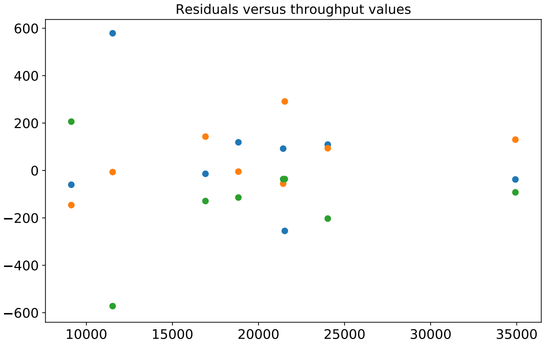
- The scatter plot of errors versus predicted response does not have any trend;
- The normal quantile-quantile plot of errors is linear;
- Spread of y values in all experiments is comparable.

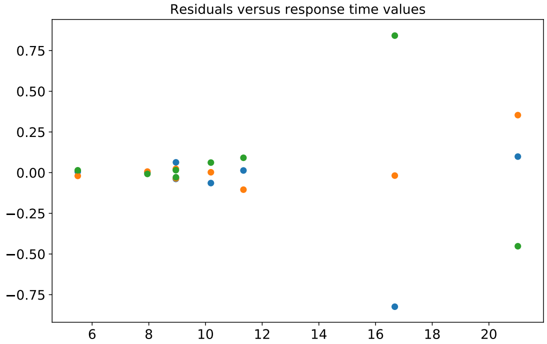
The last one is correct, y_{max}/y_{min} is not significantly different (at max around 3-4x difference).

The first two requirements illustrate following plots:





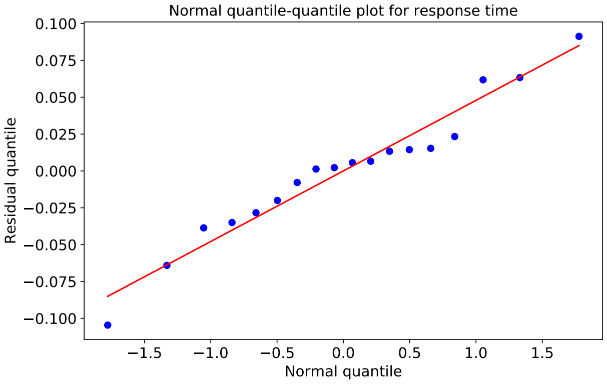




We see here that for response time there is a trend in errors, this may be due to additional latencies introduced by the network during the concrete experiments. However, since the error percentage is rather low (not greater than 4%) this could be still a valid model.

We actually also see that those relatively high errors in response time correspond to the same high errors on the throughput plot. There are couple of experiments were we generally see rather high errors, with 1 middleware - 1 server, 1 middleware - 3 servers, 8 worker threads. Apart from these two experiments we see that errors are small and comparable. Due to these high fluctuations we could suggest that these two last point are outliers of the experiment. For the throughput values, corresponding to these error metrics, we see the same - large difference from the other errors.

If we would plot the normal quantile-quantile plot without two outliers, we would see the following picture:



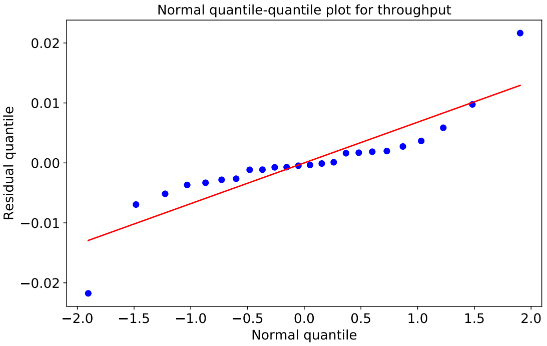
The interactive law holds for the presented values.

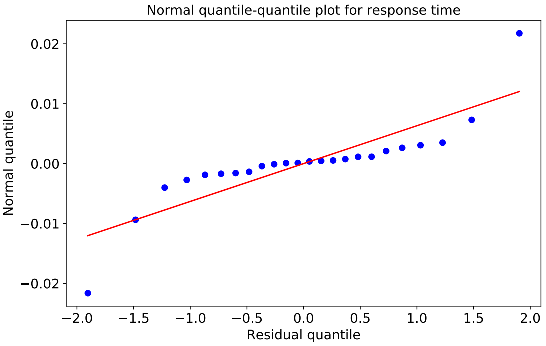
Multiplicative model So, as we saw our results are described quite good with the additive model. This is mainly on the one side because of the low impact of servers on the throughput values and on the other side, because of the fact, that for example 64 threads, both with and without middleware give almost the same numbers for the throughput as the saturation on the server side is already reached.

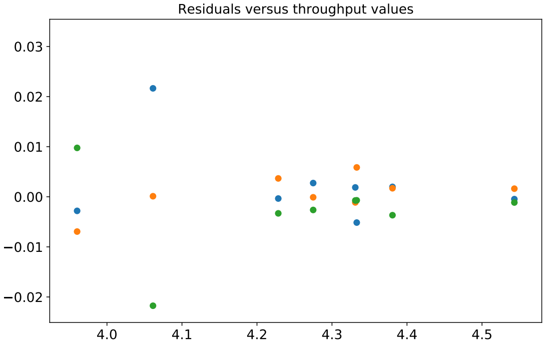
Actually when applying the multiplicative model to these results, we should receive the similar influence percentage.

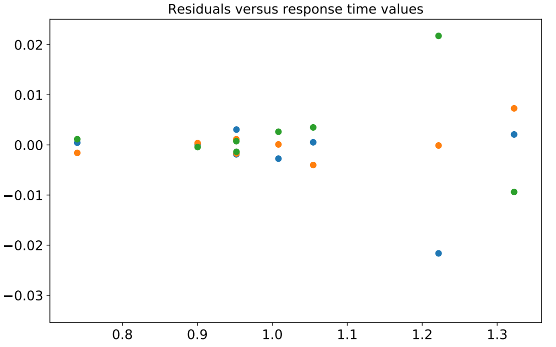
To compute the values for multiplicative model, the *log* of the y values should be computed and then the same procedures, as described above, should be applied to these new values.

The following graphs present results for multiplicative model.









We see quite the same picture. But in general, in throughput we could see now more clearly those values with high errors from the two experiments, the other errors seem to be o quite the same level.

In conclusion, we could say, that due to network fluctuations and changes in latency there were two experiments, with higher errors (the response time for them turned out to be also higher, as the system was already saturated there). This influenced on the plots for residuals, but as the errors are still rather low compared to actual results, the general trends may be identified still correctly by the model.

4.2 Throughput and Response time Impact (40 pts)

We will base our analysis on the throughput numbers, as the outliers in response time errors could influence on the values. In general, the values for the contribution of the three factors on their own are consistent over throughput and response time.

With the increase of the servers, we don't get significant impact on the performance, on the opposite side the increase in middleware or workers seems to be more beneficial according to the model.

The explanation will follow in the next section.

4.3 Comparing Model Results with Experimental Results (20 pts)

The received results correlate with the observations. The server decreases the throughput in case of 1 middleware and 8 workers, because of the increase in service time due to the addition of a slower server. In other cases it increases the throughput, as with hight number workers, the middleware could operate on the with the higher throughput and the high load from one server is then distributed across multiple servers. So, in genera the ign of the influence is positive.

Worker threads increase the throughput dramatically, which correlates with our experiments where the bottleneck were worker threads. Additional middleware influence on the throughput also positively, however the effect is less than from adding of additional worker threads.

5 Queuing Model (80 pts)

5.1 M/M/1 (30 pts)

Observations Before computing values with this model it should be said that M/M/1 makes several assumption which are not true in this configuration. Service time is supposed not to grow with the increasing load, thus it is supposed to be constant. However, this is not true in our case and the memcached server's service time grows with the increase of the number of clients. This may happen on the one side, due to some network buffering, on the other side due to the loaded memcached, which starts to be slower as the load increase.

The most important assumption which does not hold and probably due to this all timing number for the first sections will be incorrect is that M/M/1 queue essentially assumes that only one request can be processed at a single time moment. For us, for example, for 8 worker we have

5.2 M/M/m (30 pts)

The Little's law is however true for all configurations.