Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Petar Ivanov***
Legi number: ***16-931-636 number***

Grading

Section	Points
1.1	
1.2	
1.3	
1.4	
2.1	
2.2	
3.1	
3.2	
3.3	
Total	

1 System Description

1.1 Overall Architecture

The main part of the implementation was separating the functionality into suitable components each with strictly defined responsibilities. The entry point to the system is a class called **HashingServer** [1]. It is responsible for accepting connections from clients, passing requests to the next component and sending the response back to the client once it has been received from the server. In addition the HashingServer logs the timestamp data for each hundredth request that passes. This component operates in a loop where at each cycle it listens for incoming client connections and requests. The main technology used behind this process is Java NIO. Once a new client is detected, a new non-blocking socket connection is established for communicating with that client. It is then registered with the server's selector, which is notified every time a request is coming or has to be sent on that channel. This makes connection handling asynchronous because once a request has been sent for processing, the server can go back to listening for other request.

Each request is then processed by a **RequestHandler** [2]. The operation of the RequestHandler depends on two other components, which are explained shortly. When it gets the request, it has to decide which server it will be sent to and which queue it will be put in. For that a **RequestParser** class [3] is used. The RequestParser extracts the operation type (get, set or delete) and the key. The key is then passed to a **HashingHandler** [4] component which gets the hash of the key and consults an internal data structure to decide which server will handle that request. After a decision is reached, the HashingHandler returns a **QueueManager** [5] that is associated with the chosen server. The functionality of the HashingHandler is described in the next section. The RequestHandler finally passes the request to a QueueManager.

The QueueManager maintains two queues - a BlockingQueue for the get requests and a ConcurrentLinkedQueue for sets. The reason a ConcurrentLinkedQueue is used, is to avoid deadlock in the thread taking requests off the queue. When the QueueManager is created it instantiates and starts both the SET thread and all the GET threads. It then waits for incoming requests from the RequestHandler and puts each request in the correct queue. The GetThread and SetThread are the implementation of how gets and sets are handled.

Finally a **MiddlewareDataEventClass** [6] is used to store information about the client requests. First of all it stores the request data, the server response and the socket on which it was received. This is used both by the SET and GET threads to notify the HashingServer that data is ready for writing back to the client.

The following timestamps are logged and stored in each MiddlewareDataEvent: $T_{parsetime}$ (the time it takes to parse the request), $T_{hashtime}$ (the time it takes to hash key and map to server), $T_{queuetime}$ (the time spent by each request in the queue), $T_{processtime}$ (time between taking event off the queue and sending to server - relevant for SETs), $T_{servertime}$ (the time it takes to receive the response from the server), $T_{returntime}$ (time from receiving the response from the server to sending it back to the client), $T_{totaltime}$ (the total for the request), $T_{successful}$ (record whether the request was successful).

[1] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/HashingServer.java`

[2] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/RequestHandler.java`

[3] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/RequestParser.java`

[4] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/HashingHandler.java`

[5] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/QueueManager.java`

[6] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/MiddlewareDataEvent.java`

1.2 Load Balancing and Hashing

The way hashing is handled in the middleware has the biggest effect on the load balancing between servers. The high level description of how this works is first uniformly separating the statespace of possible hash values between the servers. The key of each request is then hashed and based on the interval in which the returned value falls, the request is sent to the corresponding server.

The first approach I took was dividing the statespace of Integers equally based on the number of servers 'n' and then assigning each server a specific interval. Although this ensures a fair separation, it is not an acceptable approach for a distributed system because the number of servers may vary. In case server nodes are added or removed, 'n' will change and each new request will be hashed to a different interval of the statespace. Thus remaining servers will start receiving requests, whose keys used to map to another location, thus invalidating the existing cache on that server. With this approach I also used a naive hashing algorithm for strings called djb2 hash. Although simple to implement it results into a moderately high number of collisions compared to other algorithms and a non-uniform distribution.

For the final implementation I used Consistent Hashing. The way it works is by placing all server nodes on a data structure that simulates a circle. The circle represents the statespace of possible values returned by the hash function and each node is assigned a point on that circle. Thus each server is associated with an interval which is the range of points between to nodes on the circle. In order to find which server a request should go to, the key for that request is hashed and mapped onto the circle. The server which will take the request is the first one that is found when moving in a clockwise direction. This approach achieves two properties that the naive one was lacking - when a server is added it will get the same load as the other machines and if one is removed the requests will be shared among the other servers.

However, for this to have the described properties a modification has to be made. Since the nodes are initially placed randomly on the circle it is possible for two points to end up close to each other, thus resulting in uneven interval and poor load balancing. To tackle this each node is replicated a number of times and all replicas are put on the circle as well. This results in much better load balancing as shown in my experiments. Initially I set the number of replicas to 100 and this improved the load balancing, but not by much. Setting it to 10000 replicas resulted in very even distribution.

One final detail is that the hashing algorithm used for servers and keys should be the same. The reason is that changes to the number of servers will result in requests being handled by the next server on the circle, thus spreading the load and ensuring that the change affects only a small part of the statespace.

The djb2 algorithm was also substituted for md5. It results in less collisions and more uniform distribution. As a result of using md5 the statespace was also extended to that of Long. Since md5 produces a 128-bit byte array I fill the Long variable with the first 64 bits, from which the hashed value is calculated.

1.3 Write Operations and Replication

As described in the Architecture section the SET operations are handled by a **SetThread** [7] class. For each SET queue there is a single thread that is taking requests off the queue and sending them to the server. The SetThread has one connection per memcached server. In the case with no replication it just has a single connection to its corresponding node.

Similarly to the HashingServer, the SetThread operates in a while loop. At the beginning of each iteration it tries to take a request from the queue. If it finds one then it prepares each connection for writing to its memcached server and assigns the same request to be sent on each

[7] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/SetThread.java`

one. Again in the non-replication scenario there will be only one connection and the request will be sent only on it. Then the selector is queried and responds with all connection that have an action pending. If the action is writing, the thread sends the request and goes back to handling other connections. If replication is used, then all replicas will be sent in the same iteration.

Handling read events is a little more complicated because of replication. In order to send a response back to the client, the set thread has to know when all servers have responded. This is achieved using a map from sockets to a queue of requests. Every time a request is sent to the server it is put in the queue for the socket. When a response is read on a connection, the head of the queue for that socket is polled and the the response data is added to it. This approach works because memcached sends responses in order. A counter on each request is used to measure how many servers have responded. If the number is reached then the response is passed to the HashingHandler for returning back to the client as described in the previous section. Each time a response is received it is check for successfulness and that information is recorded. A request is considered successful if all memcached servers return a "STORED" string.

If the set thread does not retrieve anything from the queue in the beginning of the iteration, it will just wait for either a read event on of the connections or a notification form the QueueManager that something is available in the queue.

A note that has to be made on the performance of writes is how the server time for a SET request is calculated. Measurements start when the first replica is sent and finish when the final server has responded to the request. This means that while the time for a single request is trivial, in the replicated case the response time will depend on the server which is slowest to send its response. Thus a scenario where 4 servers respond immediately and the fifth takes a long time is possible and will largely affect the total time for the request.

The replication time I expect to see in later experiments can be formalized mathematically. Assume that the time it takes to send a single request and receive a response for it is x. Then $T_{replication} = x + (x' - x)$, where x' is the total time for the the replica that takes the most time. Thus the final formula can be expressed as $T_{replication} = x + \delta x$. However, unless that memcached server is overloaded the longest replica will be read by the connection in the next iteration, where read operations occur, thus making $\delta x < x$. In other words I expect the replication to take no more than twice the time for a single write.

Another point is that the $T_{servertime}$ will be very different between running on localhost and on Azure. The reason is that I expect an overhead to be caused by sending data over the network. This is also where I expect a possible bottleneck, as the performance will be largely dependent on the Azure internal network. Another thing that can limit the request rate is parsing each response from the server to check if it has been successful or not. This functionality happens for all replicas and has to be executed before the request is returned to the client, therefore it adds to the total time of the request.

1.4 Read Operations and Thread Pool

Similarly to SETs, GET requests are handled by a **GetThread** [8] class. This is another example of a component that underwent a significant change since its first implementation. Initially there was a singe thread that took request from the queue and used an ExecutorService with a bounded ThreadPool to execute get requests. However, this was adding an unnecessary overhead because a thread-safe pool of connections to the memcached server had to be maintained. Every time a task was set for execution, a connection had to be taken off from the thread pool and after the thread had finished it was responsible for safely returning the connection back to the pool so that it can be used for other requests. Ultimately this was adding more complexity (lock contention) for no real gains to the system.

[8] `https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/src/server/nio/GetThread.java`

Instead of using an intermediary thread, in the latest implementation the QueueManager instantiates and starts 'n' GetThreads. Each of them is responsible for querying the GET queue itself and processing whichever request it takes off from the queue. Each thread also has its own connection to the server so it does not suffer any overhead from getting and returning connections from/to a connection pool. Ultimately these threads serve the same purpose but are isolated from each other. The only place they partially meet is when they query the queue for new requests.

This design is very similar to the producer-consumer pattern described in "Java Concurrency in Practice". One thread is placing requests in the queue and n threads are trying to take off requests one by one. Therefore, because of multithreaded access, the data structure has to be thread safe and the natural choice for the producer-consumer is using a BlockingQueue. Since the queue is unbounded, the producing thread can just place the request and go back to accepting new ones, thus uncoupling the components. The GetThreads are attempting to remove requests in a safe manner by blocking until a new request becomes available. When this happens only one thread at a time can get access to the critical zone and take the event from the BlockingQueue. It then processes it, while the others continue to wait for an event.

After getting a request each thread behaves synchronously and blocks after sending the request to the server. It then reads the response and uses the data stored in the MiddlewareDataEvent to send back the response to the HashingServer, which in turn sends it to the client. The thread then goes back to waiting for new request in the get queue.

2 Memcached Baselines

The aim of this experiment was to measure the performance of the system without the middleware. The experiment was carried using a bash script and followed the specification in the table below.

Number of servers	1
Number of client machines	1 to 2
Virtual clients / machine	1 to 64
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	microbench.tar.gz, microbench_parsed.log

The baseline experiment consisted of 128 instances - one per number of clients using the system. Each instance was executed 5 times and the results were averaged in order to get a stable measurement. The first 64 instances were run on a single VM where the number of threads was incremented at each instance. For the other 64 cases one machine was running with 64 threads and the second one's thread count was incremented at each instance. Parsing the experiment data for two machines was different because the response time and standard deviation had to be averaged between the two machines as well, while the throughput was averaged for each machine and then summed to get the correct TPS for that instance. An important point to make is that the memcached instance running on the server was reset every time between experiments in order to avoid any abnormal variance in the results caused by existing data in the cache. Another approach to designing the experiment was to run all instances on two machines and increment the number of thread interchangeably on the servers. This would separate the load between the two machines at all times, but I decided to run the experiments with the first approach because it is less trivial and could lead to more interesting results.

The baseline experiment generated 960 files which are stored in a archived file called microbench.tar.gz. The script that parsed the data stored in those files produced a single file log with 128 lines, called microbench_parsed.log

2.1 Throughput

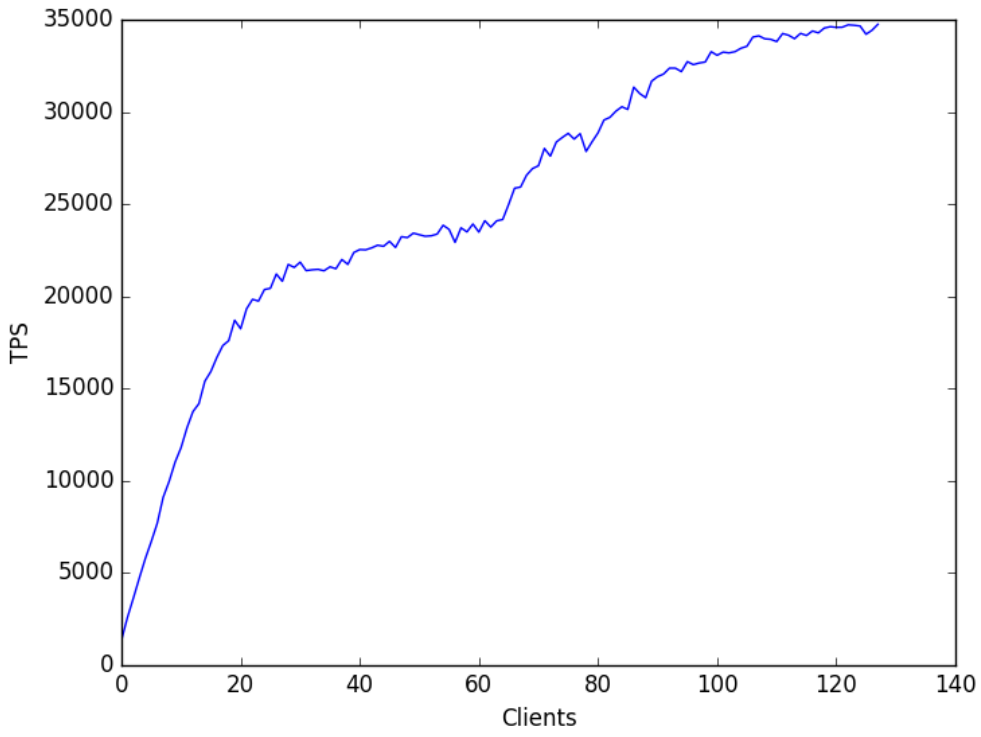
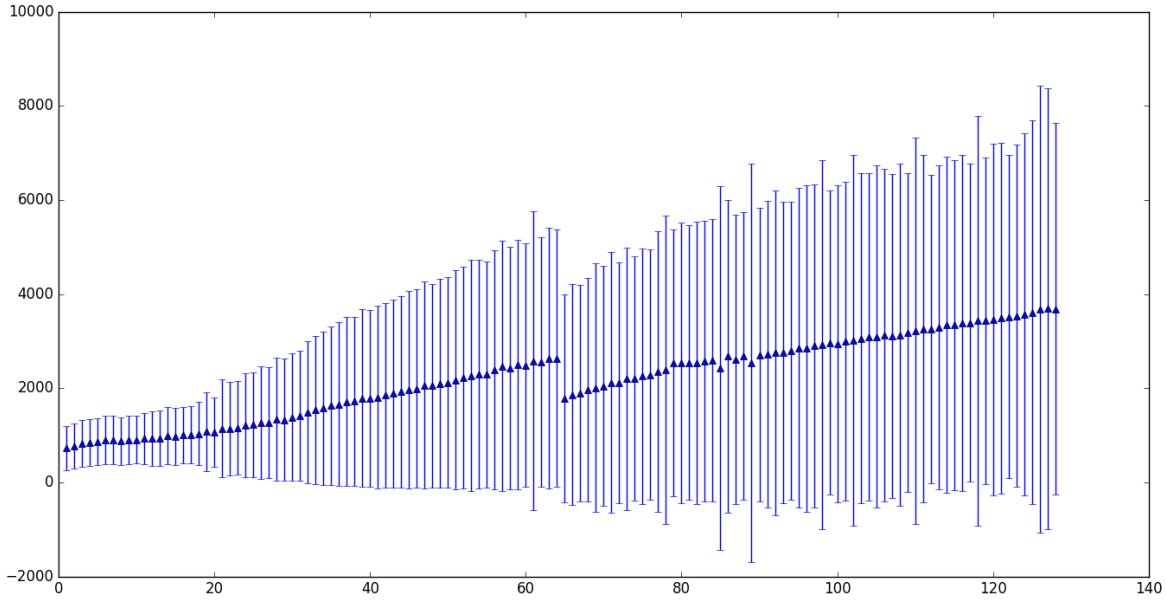


Figure 1: Middleware throughput for baseline

The results suggest that 128 clients spread across two VMs start saturating the server at around 35000 requests. Also because of the approach taken to this experiment another conclusion is that a single VMs starts reaching a steady throughput level at around 21000 requests with 64 virtual clients. The reason there will be a limit for the TPS on a single machine is that it has a limited number of cores and the threads do not actually run in parallel as they would if executed on separate machines. The client VMs which I tested, on have 2 cores, which means there is a lot of context switching between threads and there is a limit to the load they can put on a server.

As expected when a second machine starts sending requests, the throughput rises significantly as long as the number of clients on this machine get enough processor time. Just like with the first VM, the server gets saturated when the second client reaches 64 threads.

2.2 Response time



The response time graph shows the same pattern as the throughput. The interesting part to know is that, again because of the experiment approach the response time for 65 clients is actually lower compared to 64. This is because the response time for the second machine, which runs only one client has a very high response time, while on the first one, the 64 threads wait for CPU time. The results for the two machines are averaged to give this result.

3 Stability Trace

There are two types of log files as a result from this experiment. stability0, stability1 and stability2 are the log files for each of the clients and RequestLog.log is the log file containing all the timestamp information for the sampled requests. 20 seconds of warm-up time are allowed for the system so that time period is not displayed in the graphs. The experiment was run with a thread pool of 8 read threads.

Number of servers	3
Number of client machines	3
Virtual clients / machine	64 (explain if chosen otherwise)
Workload	Key 16B, Value 128B, Writes 1% (see footnote)
Middleware	Replicate to all (R=3)
Runtime x repetitions	1h x 1
Log files	stability0.log, stability1.log, stability2.log, RequestLog.log

3.1 Throughput

As seen from the graph the middleware has stable performance for the duration of the one hour experiment. The throughput may vary at times but there are several factors that affect this. For example when the garbage collector kicks in this may result in a local minimum in the graph. The effect of replication to all servers is also visible on the graph. In the iteration where a request is sent to the server, there will be no reading for responses. This is because I am using a single read/write connection to each server and it will be busy writing. At the end there is a spike in the performance, which is due to a very small window where one of the clients stop but the others continue working for a one or two seconds more. This is the cool-down phase and

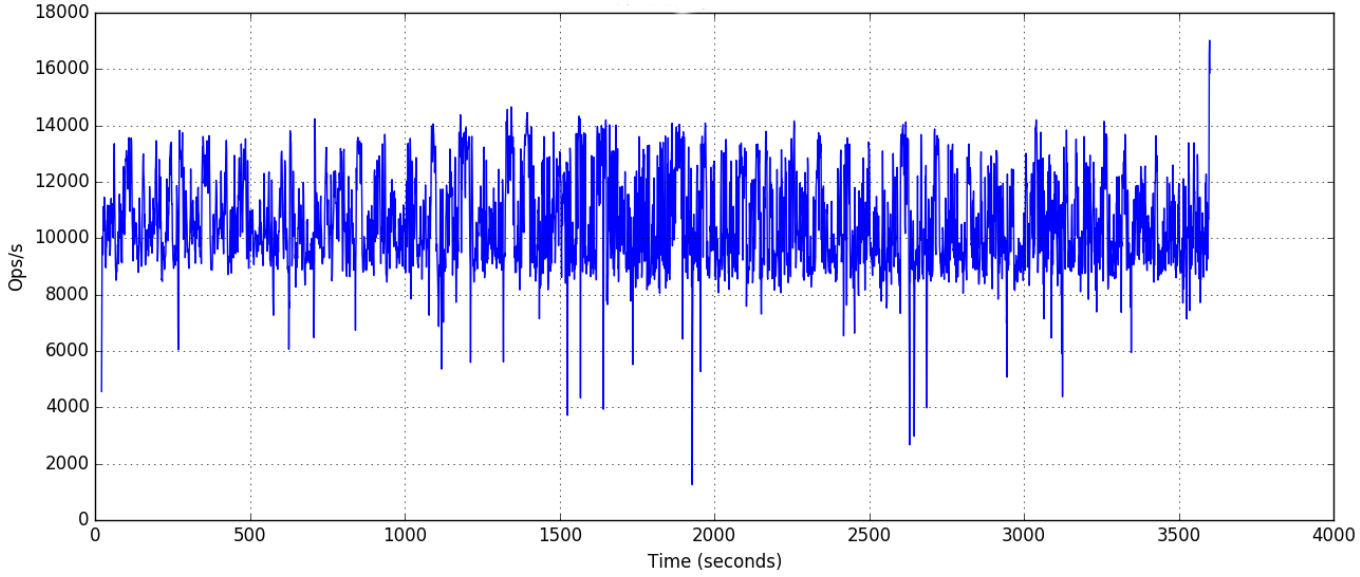
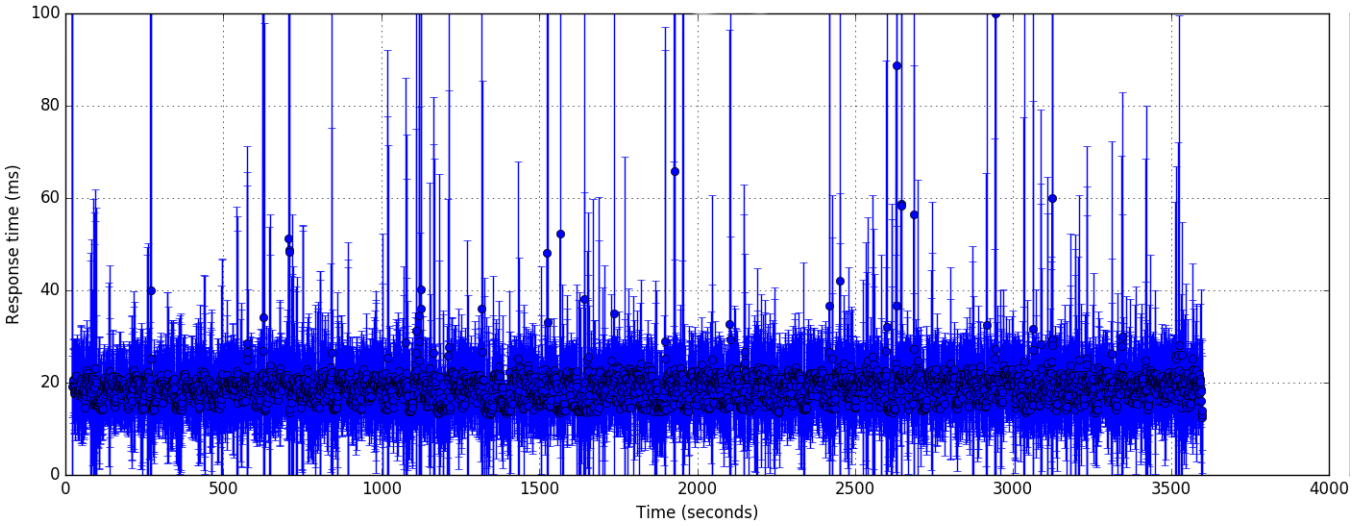


Figure 2: Aggregated Middleware throughput

it was intentionally left on the graph, in order to show there will be a change of performance depending on the clients.

3.2 Response time



The same applies for the response time as well. Highs in the response time correspond to lows in the throughput. In other words, if the response time is high at a certain interval, then the throughput will be lower.

3.3 Overhead of middleware

Before explaining the overhead which the middleware causes in its final implementation, I think it is worth mentioning how the performance was affected by different factors throughout the development process. The last week before the deadline was spent on debugging an issue that caused performance to drop over time. This debugging process lead to several findings that improved performance and reduced the overheads.

The most important thing to mention is that the issue was caused during a certain stage of request handling. After logging the timestamps explained in Section 1.1, I discovered that

the problem is caused by the server time for SET requests as it can be seen from the graph below. The total time for the SET request grew asymptotically with the server time for SETs (the other timestamps add very little overhead compared to $T_{servertime}$).

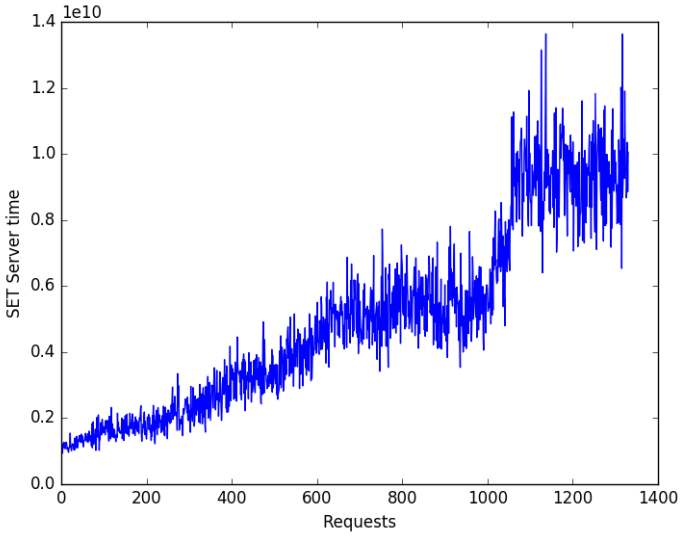


Figure 3: SET server time before optimizations and fixes

That is the time between sending the request and receiving the replies from all replication servers. All other timestamps in the middleware are stable and there is no increase of the response time.

The discoveries which I made during the debugging process are:

- RequestHandler was not putting equal load on all servers. This was caused by using a hash function which was too simple and the statespace of all integers for the nodes and intervals. After fixing that as described in Section 1.2, the performance as a whole improved, but there was still a drop.
- Allocating a new buffer every time when a SET request had to be sent to the server is wasteful. That is why this was substituted by using a single buffer for all write requests, which saves memory.
- Memcached servers return a positive response to all requests, which means that requests are not being dropped.
- Not doing any replication resulted in the same performance, which meant that the way replication is implemented was not the problem.
- The performance drop was much smaller when the program is used on localhost.
- If SETs are handled synchronously, i.e execute next set request once all servers have responded to the last, removes the problem. However, this is not a solution, but helped in identifying the problem.

There were very few changes required to make the sets synchronous, which means that the domain where the problem lies is now very little. Finally I discovered that it was possible to get several responses from memcached in a single read operation on the corresponding socket. In that case only one client request would receive a response, and the others would be left waiting. Since it is a closed system, that client would just go on waiting forever and not sending new

requests, which resulted in the steady drop of performance. To fix that I reimplemented the response verification function that is executed every time a read occurs.

Even before doing any experiments it was clear that when using a middleware, the throughput will be lower and the response time - higher. This is because it does a lot of computation that is not present when a client is communicating directly to a server. However, I expect that the middleware will compensate for that in scenarios where several clients are overloading one server, while others are not performing tasks because there is no component to distribute requests. However, for the time being I will focus on the latencies that exist at the middleware.

Using the baseline statistics, which I obtained previously, and the stability traces, there are a number of conclusions that can be made for the performance of the middleware. First of all, because of the approach taken to the baseline experiment, I saw that a single machine with 64 threads reaches saturation at around 21000 TPS. This meant that a single machine using the middleware would get less TPS. Without having run the stability trace I expected that a single machine with middleware would get between 10000 and 15000 TPS. Indeed, this is not too far away from the truth because the mean of the stability trace for 3 client machines is around 11000, which means a single client would get around 3500. This number I expect to be 3 - 4 times higher when there is a single client using the middleware.

As seen from the graphs the response time is also higher. A single client, running 64 threads, gets a response time of around 2000 nanoseconds. The mean response time for 3 clients using the middleware is 20 milliseconds. Again a single client using the middleware will get a lower response time which I expect to be less than 5 milliseconds.

The response times are best compared using a table that presents the overheads of the middleware. The numbers used for that comparisons are extracted from the RequestLog.log file, which contains the timestamp data collected during the stability trace. The timestamp values in the table are samples that provide and intuition of the overhead.

Overhead	Middleware	No middleware
Network	$2 \times t$	t (2 send/receive per request)
Request parse	1000 ns	N/A
Hashing	2200 ns	N/A
Queue time	1698300 ns	N/A
Server time	9407000 ns (incl. replication)	Single response
Server time (formal)	$x + \delta x$	x
Response parsing	66100 ns	N/A
Returning response	2661000 ns + $T_{network}$	$T_{network}$
Logging	T_{log}	N/A

A final point to make is that logging is one of the processes that provides the largest overhead, because writing to a file is an expensive operation. To minimize this there is a sampling rate for every hundredth GET and every hundredth SET request. Furthermore, to avoid writing to several files, which will be very messy, logging is done only on the HashingServer thread, before sending the response back to the client. This approach will not suffer from wasting time on waiting for a lock to write to a file, because logging is done only from one thread. However, from now on logging will not be considered as overhead of the middleware, because this is an essential part of doing performance analysis and is not related to request processing.

Logfile listing

Short name	Location
microbench.tar.gz	https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/mic
microbench_parsed.log	https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/mic
stability0.log	https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/sta
stability1.log	https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/sta
stability2.log	https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/sta
RequestLog.log	https://gitlab.inf.ethz.ch/ivanovpe/asl-fall16-project/blob/master/Req