Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Lukas Burkhalter***
Legi number: ***12-915-914***

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

In this section, we first outline the overall architecture of the load-balancing system for memcached, followed by a subsection about the load balancing method. Finally, we conclude the section with a detailed description on how reads and writes are handled in the middleware,

1.1 Overall Architecture

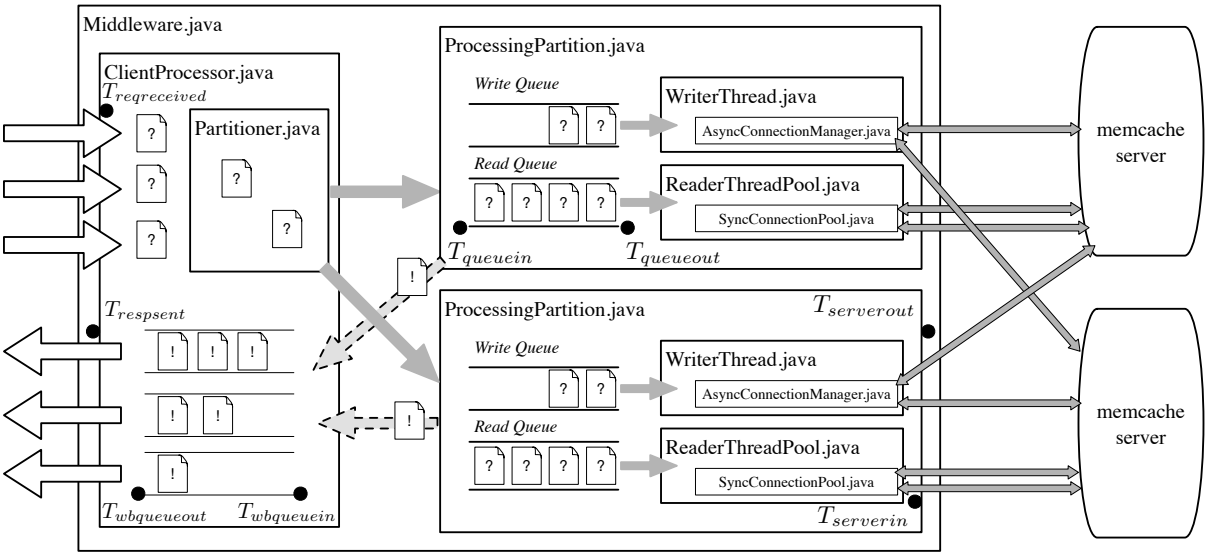

Figure 1: An architectural overview of the load-balancing system. The locations of the timestamps are marked with a black dot with their corresponding name.

A high-level overview of the middleware implementation is depicted in Figure 1. The provided abstract architecture provided in the lecture is implemented in the *Middleware* [1] class. Each request from the clients is handled by the *ClientProcessor* [2], which implements the network interaction with non-blocking I/O using Java NIO [2]. The class supports one thread for accepting, reading and processing requests and up to n writer threads for writing responses back to clients. After receiving and parsing a request, the request is forwarded to one of the partitions based on the load-balancing algorithm implemented in the *Partitioner.java* [3] class.

A partition is abstracted in the *ProcessingPartition.java* [4] and is responsible for forwarding request to the corresponding memcached server. Each partition has one write queue and one read queue, which are processed either by the writer thread implemented in *WriterThread.java* [5] or by the reader thread pool implemented in *ReaderThreadPool.java* [6] respectively. The writer thread uses non-blocking I/O for communication since he has to replicate write request on multiple memcached servers. A writer thread in a partition has access to one *AsyncConnec-*

[1] https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/Middleware.java

[2] https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/ClientProcessor.java

[3] https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/messages/Partitioner.java

[4] https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/ProcessingPartition.java

[5] https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/WriterThread.java

[6] https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/ReaderThreadPool.java

tionManager.java [7] that manages an asynchronous connection to each memcached server. In contrast, read requests are processed by multiple reader threads, which are using traditional synchronous connections for communication. The reader threads in the pool share a *SyncConnectionPool.java* [8], which maintains up to n connections to one memcached server, for accessing connections. After finishing a write or a read job, the resulting response is not directly sent back by the processing thread. Instead, the processing thread enqueues the response in writeback queue located in the client processing unit for leveraging the benefits of non-blocking I/O. The back-writer threads in the client processor are working on the writeback queue for sending the responses back to the clients. Summarized, the main design decisions are:

- Non-Blocking I/O on the client processing side and in the partition writer thread.
- Blocking I/O for read requests combined with multiple reader threads.
- Processing threads in the partitions do not directly send results back to the client. Results are queued in a writeback queue and processed by client writer threads.

1.2 Load Balancing and Hashing

In order to distribute the client requests among the partitions, each request key is mapped to a partition id by a load balancing function. Our system implements the function by first using a hash function that maps the key to an integer space. The integer space is divided into segments of equal length and each segment corresponds to one partition. By determining the segment in which the integer hash is located, we get the id of the resulting partition. We are implementing it in the following way, where it maps a key k to an id in $[0, n)$ with a hash function H, where n is the number of partitions and N the size of the integer space.

$$id = \left\lfloor \frac{H(k)}{\lfloor N/n \rfloor} \right\rfloor \; mod \, n$$

Instead of directly taking the modulo from the hash, we divide by the interval size to determine the id from the interval and compute the modulo only for the corner cases. Therefore, we avoid only considering the first bits. In order to guarantee a uniform load distribution with this approach, a hash function that produces good pseudorandom values is required. In our implementation, we use a 32-bit cyclic redundancy check (CRC-32) function as the hash function. CRC-32 is originally intended to use as 32-bit checksum in codes. However, it has been shown to perform well on load-balancing task as it is fast and produces pseudorandom numbers as it depends on xor [3, 4]. In comparison to cryptography hash functions such as MD5, CRC-32 is faster and suffices for the task. The implementation can be found in the *IntervalCRC32Hasher.java* [9] class.

1.3 Write Operations and Replication

When a write request enters the system, the system first determines the partition on which the request should be processed based on the load balancing algorithm. If the partition is selected, the write request is enqueued in the selected partition's write queue. The writer thread of the partition (*WriterThread.java* [10]) works on this queue piece by piece. For each write request

[7] `https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/connection/AsyncConnectionManager.java`

[8] `https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/connection/SyncConnectionPool.java`

[9] `https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/util/IntervalCRC32Hasher.java`

[10] `https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/WriterThread.java`

the writer hast to forward the request to up to n memcached servers, because the middleware also has to support replication. Therefore, the writer has to write up to n-times more data to the network in a replicated write request than in a non-replicated write request. Since the write request can contain a lot of data, the replication cost even more. Furthermore, each request hast to be sent over a different connection. For avoiding waiting on connections during writing and reading, our writer thread uses non-block-I/O to send and receive data from all the replicating servers and works on all channels simultaneously. However, a replication writer has to write n-times more data and has to wait for n responses in contrast to a single-location writer. The latency for a single write would be the time to put a request on the wire plus the waiting time, which includes the round-trip time to the server and the server processing time. In the replication case, the latency for putting the request on the wire is n times higher as there is n-times more data. However, it might be possible to use the waiting time for further writing or reading from other channels. Therefore, the waiting latency is not n-times higher than in the single-write case, it can be reduced with non-blocking I/O. As a result, the rate that will limit the writes that can be carried out is the network bandwidth or the maximal rate at which the system is able to write to the network

1.4 Read Operations and Thread Pool

In contrast to the write operations, read operations only have to be forwarded to one memcached server, which belongs to the partition. Similar to the write request, a read request is first assigned to partition by the load balancing algorithm and then enqueued to it's corresponding read queue. The reader thread pool (*ReaderThreadPool.java* [11]) containing of several reader threads works on this queue on a piece by piece basis. Since multiple threads access the same data structure, some sort of synchronization has to be performed in order to avoid race conditions and unwanted states. In our implementation, we use Java's LinkedBlockedQueue [1], which is thread safe, for the read queue. Each reader thread processes one request at the time and forwards it to the memcached server. As only one connection is involved, we use standard synchronous java sockets for the communication. For sending the request, the reader first has to acquire a connection from the connection pool (*SyncConnectionPool.java* [12]), which manages up to n-connections to the corresponding memcached server. If less connections than threads are available, they compete against each other. As the work of a reader thread mainly involves writing and reading from a connection, it might be a good idea to chose the number of connections equal to the number of threads as the threads to communicate synchronously and wait for the response. Else a lot of time would be spent on acquiring a connection, and connections would be blocked still if the owner thread is waiting on the connection. After the read task is finished, the response is enqueued in the write-back queue.

2 Memcached Baselines

To evaluate the basic performance measures of a memcached server without our middleware in front, we conducted an experiment by varying the number of clients. We focused on the aggregated throughput and response time during a period of 30s. We deployed three basic A2 machines in the azure cloud. Two machines were used for creating the load with memaslap whereas one machine was running a memcached server with one thread. We varied the number of clients in both load generating machines symmetrically. Each machine started with one virtual client, the number was increased in steps of 2 up to 24 and afterwards by 4 until 64 clients were

[11] `https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/middleware/ReaderThreadPool.java`

[12] `https://gitlab.inf.ethz.ch/burlukas/asl-fall16-project/blob/master/src/ch/eth/lubu/connection/SyncConnectionPool.java`

reached. This procedure was repeated five times. For a client setup, we measured a 60s trace with 1s granularity to also consider start-up and cool-down slowdowns. The experimental setup is summarized in the following table.

Number of servers	1
Number of client machines	2
Virtual clients / machine	1 to 64
Workload	Key 16B, Value 128B, Writes 1% [13]
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	baseline.zip

2.1 Throughput

Figure 2 depicts the aggregated throughput over 30s for a varying number of clients for the baseline experiment. Until 24 clients the throughput grows rapidly as the mechached server has free resources and can process the request without delay. After 24 the throughput starts to grow linearly but with much smaller step size than before. After 88 the throughput starts to flatten and becomes more unstable but still increases slowly. With this observations, there is an indication that the one thread memcached server starts to becoming saturated after 24 clients, which is the point where the clients start to wait for each other and more clients increase the throughput less than before. The memcache server has to queue the requests and request begin to be delayed Therefore, we should observe also increasing response time after the 24 clients mark.

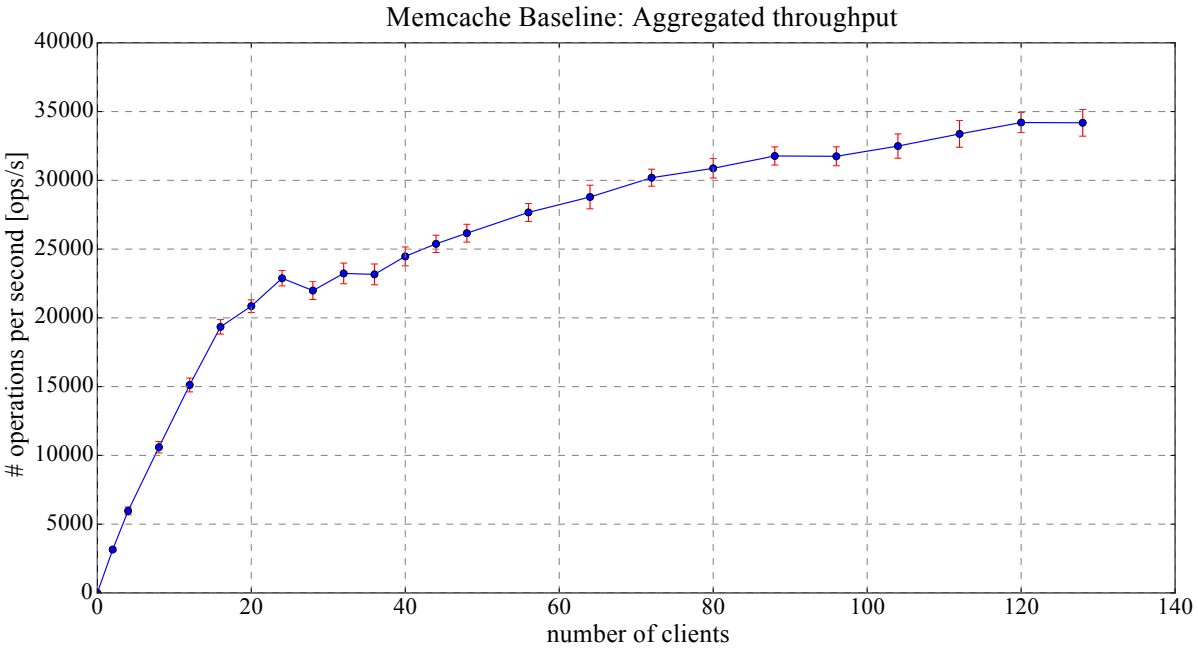

Figure 2: Depicts the aggregated throughput with increasing number of clients in the baseline experiment.

[13] As starting point use the workloads provided in `http://www.systems.ethz.ch/sites/default/files/file/asl2016/memaslap-workloads.tar`. Use by default the *small* workload. In later experiments you can and should change read-write ratios and potentially use other value sizes.

2.2 Response time

Figure 3 shows the average response time over 30s for a varying number of clients for the baseline experiment. Up to 22/24 clients, the response time stays more or less the same and is only slowly increasing. After that point the average of the response times grows linearly and the standard deviation increases. Therefore, the server starts to be saturated after this point, as the memcached server cannot handle the load anymore without delays. Also the increase standard deviation is caused by the higher delays the sever, which results in more varying response times for the clients.

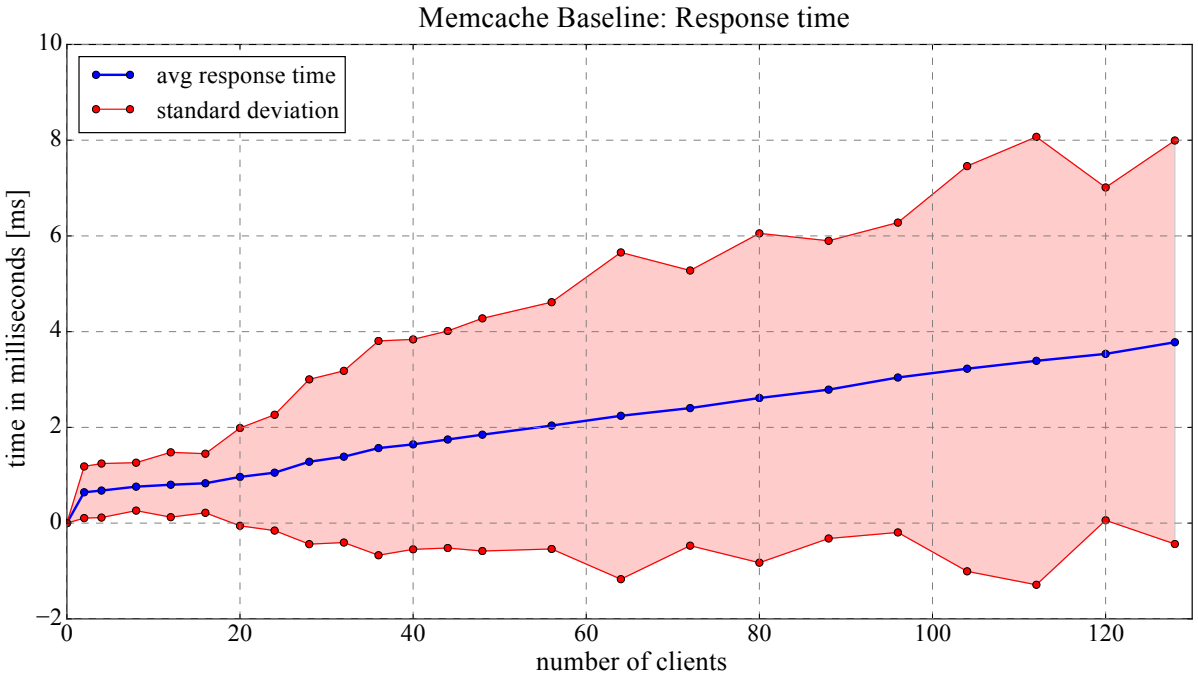

Figure 3: Depicts the average throughput with increasing number of clients in the baseline experiment.

3 Stability Trace

To evaluate the stability of our system over a longer timespan, we measured aggregated throughput and average response time over a 60 minute period. We conducted the experiment on the azure cloud with three load generating basic A2 machines, three memcached servers on three basic A2 machines and one basic A4 machine for the middleware. The memaslap load generators run with 64 virtual clients each and the memcached servers operate on one thread. The middleware has 16 reader threads with 16 sync. connections and 1 writer thread per partition, 1 client read thread and 2 client write-back threads. The replication factor has been set to 3. The experiment run for 70 minutes with log granularity 1s for also considering start-up and cool-down slowdowns. The middleware runs on a oracle java 8 JVM, without any flags. The experiment parameters are summarized in the following table.

3.1 Throughput

Figure 4 summarizes the aggregated throughput results of the long-run experiment aggregated to 1 minute. The throughput is relatively stable and lies between 12000 and 14000 operations per second. There is small down peak at the 44 minute mark, which might be caused by either network congestion or the garbage collector in the middleware.

Number of servers	3
Number of client machines	3
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 1% (see footnote)
Middleware	Replicate to all (R=3)
Runtime x repetitions	1h x 1
Log files	longrun.zip

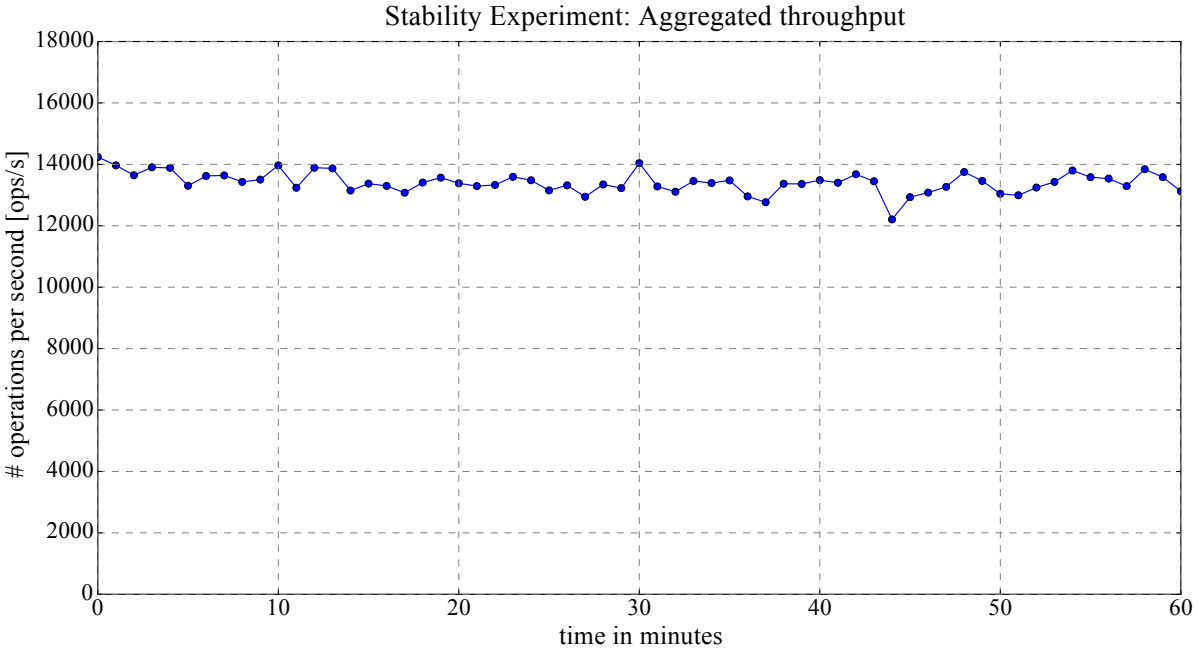

Figure 4: Depicts the aggregated throughput per minute during a 60 minutes experiment running with three client and three memcached machines and one middleware.

3.2 Response time

Figure 5 depicts the average response time over one minute during the long-run experiment. The average response time is stable around 13ms, except three peaks with higher average response time. The higher peak is also observable in the throughput plot with lower throughput around the 44-minute mark. As previously mentioned, we are suspecting network congestion or garbage collection might be the cause. Another observation is that the standard deviation is high and often goes below zero. This is an indication that the average/std model does not fit well to the data as the response time has big variations. Therefore, median/quantil might fit better to the data. The response time has big variations as a lot of latencies are involved like queue waiting time, middleware processing, network and memcached server.

3.3 Overhead of middleware

Based on the baseline experiments, we expected that the throughput with three client machines and three equally loaded memcached server would be much higher than in the baseline. With a perfect load-balancer without latency, we expected three times the throughput from the 64 clients mark in the baseline experiment, which is about 90000 ops/s in total. However, in the longtime experiment, we only measure a throughput of 13500 ops/s for 192 virtual clients, which is substantially less than expected. In the baseline such a throughput was measured with 16 virtual clients, which is disappointing for our system. If we look at the response time plot, we find the reason for the observed low throughput. The response time is about 13ms in average,

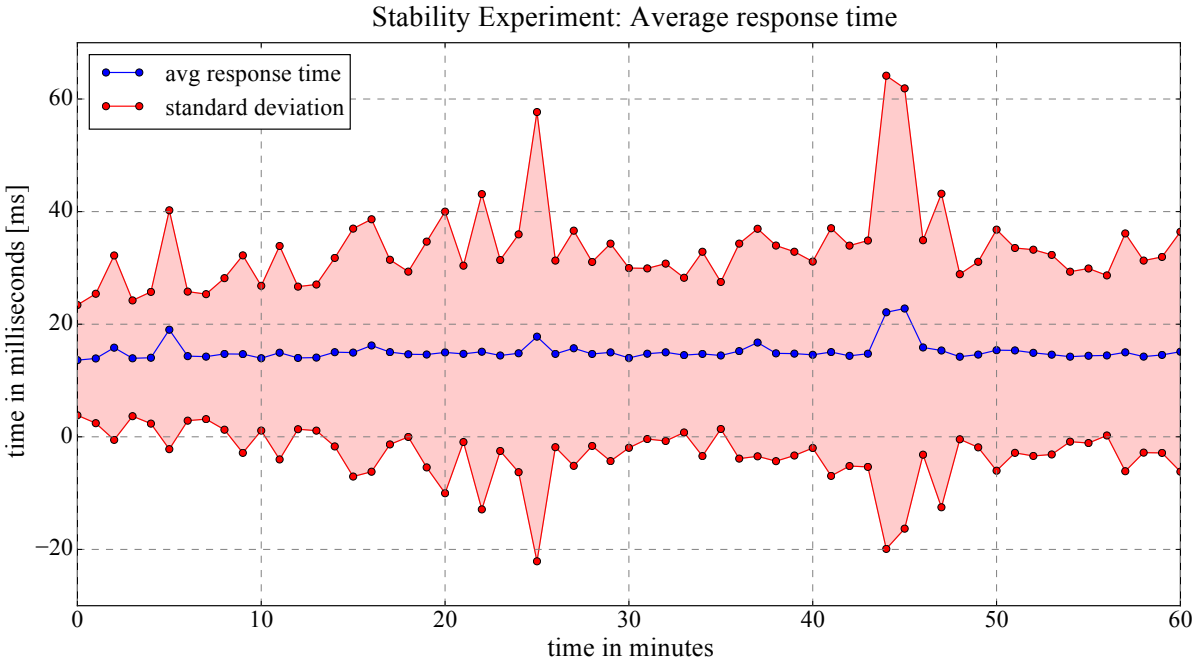

Figure 5: Depicts the average response time in the long-run experiment.

which is about 3 times higher than the response time measure with 128 clients in the baseline. Therefore, the middleware introduces a latency overhead for each submitted request by a client. We summarized the latency overheads of the middleware in the following table.

Overheads	**Description**
Network stack	A request and response additionally has to go twice trough a os network stack in the middleware, for reading and writing to the channel
Processing time	The time middleware uses for parsing and load-balance a request/response.
Replication	Writes cost more as they have to be replicated to multiple servers, even more network stack costs.
Queue time	A request has to wait in a queue in the middleware until it is processed.
Routing cost	Dependent on the network topology and middleware location, a request has to travel more distance and routers than in in a non-load-balancing case.

Since each request has to traverse the middleware, the request additionally has to move twice trough the os network stack, when the middleware reads it and writes it back. The same holds for the responses. Furthermore, the middleware performs various operations on the request like parsing and hashing, which also adds to the latency. As discussed in Section 1.3, the replication has a high impact on the write request's latency, because more data comes out than into the middleware. Each request is enqueued in the middleware, and, therefore, has to wait for processing dependent on the current load. Another aspect is routing, depending on the topology and location of the load balancer, the request packets might have to be routed differently over more hops or over longer distance, which also has an impact on the latency of a request, Summarized, the middleware introduces various latency overheads, which results in longer waiting time for clients. For future experiments, it would be interesting to see how the response time and throughput change with increasing number of clients using the middleware and which parts of the middleware have the highest impact on the latency overhead.

Logfile listing

Short name	**Location** `https://gitlab.inf.ethz.ch`
baseline.zip	`/burlukas/asl-fall16-project/blob/master/data/baseline.zip`
longrun.zip	`/burlukas/asl-fall16-project/blob/master/data/longrun.zip`

References

[1] Java linkedblockedqueue. `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html`, note = [Accessed: 2016-19-10].

[2] Java nio. `https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html`, note = [Accessed: 2016-19-10].

[3] Z. Cao, Z. Wang, and E. Zegura. Performance of hashing-based schemes for internet load balancing. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 332–341. IEEE, 2000.

[4] S. Sanguanpong, W. Pittayapitak, and K. Koht-Arsa. Comparison of hash strategies for flow-based load balancing. *International Journal of Electronic Commerce Studies*, 6(2):259–268, 2015.