Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Tom Sydney Kerckhove***
Legi number: ***15-908-064***

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 class `RunMW`[1] creates a single object of class `Middleware`[2]. It binds to a socket on the given `InetAddress`[3] and readies to accept connections using a completionhandler of class `AcceptCompletionHandler`[4]. For every accepted connection, the initial read happens asynchronously and is completed with a shared completionhandler of class `InitialInputCompletionHandler`[5]. When the first piece of data is available, it is parsed into a `Request`[6] by the `RequestParser`[7]. The `Request` and the `SocketChannel`[8] for the connecting client are bundled in a so-called `RequestPacket`[9]. Next, the request's key is hashed to an `int`. The hash modulo the number of back-end servers is used as the index of the primary server for that key. The secondary servers are the subsequent servers, up to the replication factor. If the request is a write request, a so-called replication counter is set to the replication factor in the `RequestPacket`. Then the `RequestPacket` passed on to the `ServerHandlers`[10] corresponding to the chosen servers. Each `ServerHandler` contains a `ServerReadHandler`[11] and a `ServerWriteHandler`[12].
In the case of a read request, the `RequestPacket` is added to a `BlockingQueue` in the `ServerReadHandler`. A pool of `ReadWorkers`[13] dequeues the `RequestPackets` one by one and synchronously queries the back-end server and simply passes on the response to the client.
In the case of a write request, the `RequestPacket` is added to a `BlockingQueue` in the `ServerWriteHandler`. A single `WriteWorker`[14] continuously sends all available requests to the server and adds the `RequestPackets` to a queue of sent requests. Whenever any data is available, a `ReadCompletionHandler`[15] splits it into the different responses and as many `RequestPackets` as there are responses are dequeued from the 'sent' queue in the same order. Each of the corresponding `RequestPackets` has their replication counter decremented. If a replication counter reaches `0`, the response is forwarded to the appropriate client.
An overview of the lifetime of a request can be found in figure 1
We define seven timestamps for each successfully processed request as marked in figure 2.

1. $T_{received}$: The initial input has been received by the middleware.
2. T_{parsed}: The initial input is parsed into a `Request` object.
3. $T_{enqueud}$: The `RequestPacket` is added to the appropriate queue.
4. $T_{dequeued}$: The `RequestPacket` is dequeued for processing.
5. T_{asked}: The first request is sent to a server.
6. $T_{replied}$: The final response is received from a server.
7. $T_{responded}$: The final response is forwarded to the client.

[1] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/RunMW.java
[2] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/Middleware.java
[3] http://docs.oracle.com/javase/8/docs/api/java/net/InetAddress.html
[4] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/AcceptCompletionHandler.java
[5] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/AcceptCompletionHandler.java
[6] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/request/Request.java
[7] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/request/request_parsing/RequestParser.java
[8] https://docs.oracle.com/javase/8/docs/api/java/nio/channels/SocketChannel.html
[9] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/request/RequestPacket.java
[10] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/ServerHandler.java
[11] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/ServerReadHandler.java
[12] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/ServerWriteHandler.java
[13] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/ServerReadHandler.java
[14] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/ServerWriteHandler.java
[15] https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/asl/src/ch/ethz/asl/ServerWriteHandler.java

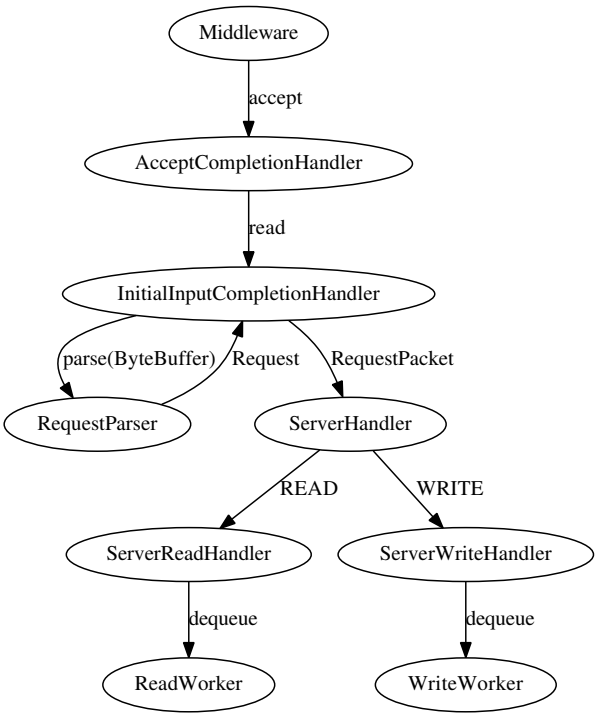

Figure 1: Lifetime of a request

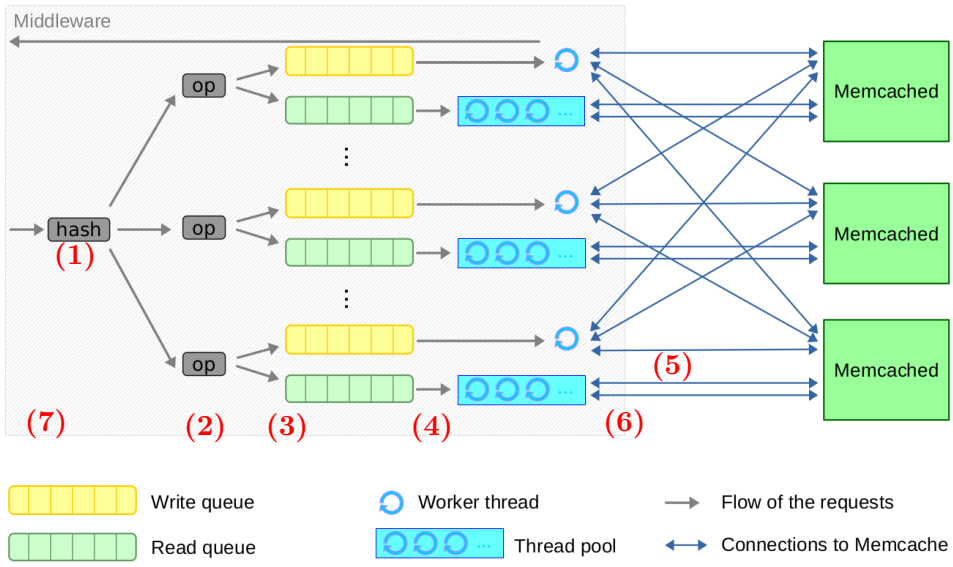

Figure 2: Instrumentation

1.2 Load Balancing and Hashing

All incoming data from the client-side is first parsed into a `Request` object. Each `Request` object contains a byte array that holds the request's key. Java's built-in function `Arrays.hashcode` is then used to hash the contents of this byte array.
Assuming that `Arrays.hashcode` distributes the results evenly among the integer domain, which is to be expected from a hash function (but not specified in its documentation [1]), the rest modulo the number of servers will be evenly distributed among the indexes of the servers.
The primary server is selected by the using the rest modulo the number of servers of the hash of the key of the request. The secondary servers are then selected by taking the next $R - 1$ servers (by index), where R is the given replication factor.

1.3 Write Operations and Replication

When a request is parsed and the key is hashed, the first discrimination on the request kind happens. Each `RequestPacket` contains an integer called the replication counter. It represents the number of servers that this request still has to be written to. In the case of write requests, this counter is set to the replication factor. Next, the entire `RequestPacket` is handed to each of the chosen servers' `ServerHandler` and subsequently their `ServerWriteHandler` where it is put on a queue, waiting to be processed..
A single thread per `ServerWriteHandler` continuously performs a simple routine: It takes one request off the queue, sends it to the server and puts the request on a so-called sent-queue. This queue holds all the requests that have been sent to the server, but for which no response has been received yet.
Meanwhile, a so-called `ReadCompletionHandler` continuously reads input from the server connection and splits the received packets of data into individual responses. For each response, a `RequestPacket` is taken off the sent-queue. The `RequestPacket`'s replication counter is decremented. When the replication counter reaches zero, all writes have been performed at the server side. In that case, the response is forwarded to the client.
When the middleware is configured to not replicate any write requests, the difference in latency between using a middleware and not using a middleware consists of 1. the time it takes for the middleware to process the request and 2. the extra network delay added by sending the request across the network a second time.
When writes are replicated, the time it takes to process a requests will most-likely increase because now it needs to wait on more than one queue to be sent to the server. The extra network delay will also increase because now the middleware has to wait for responses from all of the involved servers before it can respond to the client.
The limiting factors for carrying out writes include the replication factor and the total input to the middleware.
Because of the extra processing times and network delay, a higher replication factor should imply a lower write rate. This is assuming that threads cannot work perfectly in tandem, or that servers don't respond perfectly equally quickly.
As the total number of requests in to middleware increases, it will take more time for the middleware to read and parse each request sequentially. Similarly, each request will spend more time in the respective queues, waiting to be handled.
Network delay should not directly influence the rate at which writes can be carried out because of the asynchronous nature of handling the requests.

1.4 Read Operations and Thread Pool

A single thread in the middleware accepts connections, reads and parses all incoming requests. After the request key has been hashed, and the `RequestPacket` has been passed to the ap-

propriate `ServerReadHandler` through the appropriate `ServerHandler`, it is put onto the `ServerReadHandler`'s queue. This queue holds all requests that still have to be handled with respect to the `ServerReadHandler`'s corresponding server. The queue is a `LinkedBlockingQueue`, which is documented to be thread-safe. [2].
A pool of threads contains exactly the same amount of so-called `ReadWorker`s that are started at the startup of the middleware. Each `ReadWorker` connects to the server up front, and then starts looping through the following procedure. First it picks a single request off the `ServerReadHandler`'s queue. Then it sends this request to the server, and waits for a response. When the response arrives, it forwards the response to the appropriate client. All the input and output is done synchronously.

2 Memcached Baselines

2.1 Setup

In the baseline experiment, the idea is to measure the performance of Memcache under a typical load generated by Memaslap.
The setup is as follows. A single server runs Memcache and either one or two clients connect to it to measure throughput and response time. This is repeated for different configurations in which each client uses up to 64 clients. Each run lasts 30 seconds and the experiment is repeated 5 times for each of these configurations, for a total of 140 runs.
In each run, the throughput (transactions per second), the average respsonse time and the standard deviation of the response time are measured.
Each run represents a single line in the combined results file: baseline. A summary of the setup information can be found in figure 6.

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	baseline

Figure 3: Baseline experiment details

2.2 Throughput

The throughput in function of the total number of virtual clients is plotted in figure 4. Note first that the throughput is consistently higher when using two clients than when using a single client. This is because on a single machine, there will be more contention than when the virtual clients are divided amongst two machines. Secondly, throughput increases as the number of virtual clients increase. This makes sense as more virtual clients send more requests and ensure that queues less frequently empty. Thirdly, eventhough throughput increases, it does not increase indefinitely nor linearly. The graph clearly shows diminishing returns beyond 12 virtual clients. The throughput levels off at around 27000 transactions per second in the case of one client machine and around 40000 in the case of two client machines.

2.3 Response time

On the graph of the average response time in figure 5 we find related results.

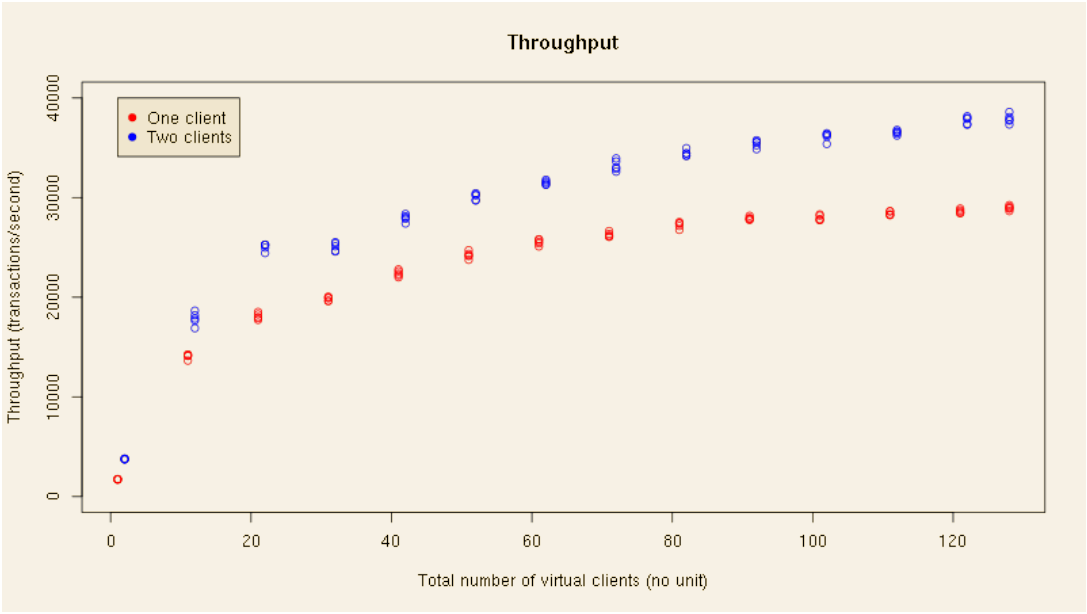

Figure 4: Throughput in the baseline experiment

First, the response time is consistently higher when using two client machines than when using one client machine. This is to be expected for the same reason as the higher throughput for two clients: thread contention.
Second, the average response time stays constant up to 12 virtual clients, and the standard deviation remains small as well. Under 12 virtual clients, the middleware is not saturated yet. Queues remain mostly empty as requests are processed faster than they arrive. Beyond 12 virtual clients, the average response time increases dramatically and linearly while the standard deviation is much higher but remains constant. The dramatic increase in response time occurs because requests arrive at the middleware quicker than the middleware can process them, so the queues rarely remain empty. Standard deviation is consistently higher beyond 12 virtual clients because the response time becomes unpredictible if requests need to spend time in a queue before they are processed. The fact that the average response time grows linearly in terms of the number of virtual clients is an immediate consequence of the design decision that queues are unbounded, that the system is closed (clients wait for a response before sending another request) and that the limit of capacity of the machines' main memory is not reached yet.

3 Stability Trace

3.1 Setup

In the stability trace experiment, the goal is to show that the middleware is stable and to measure the overhead of using it.
In order to do this, the middleware has run for one hour with full replication connected to three Memcache servers and three load generator clients.
The throughput and average response time are measured in each client every second. The results can be found in the logfiles stability-client0 stability-client1 and stability-client2. The middleware's log of the experiment can be found in stability-middle.

3.2 Throughput

The graph in figure 7 shows the throughput in each client for each second of the experiment's run time. It is clear from the graph that the middleware is very stable. It is however also very

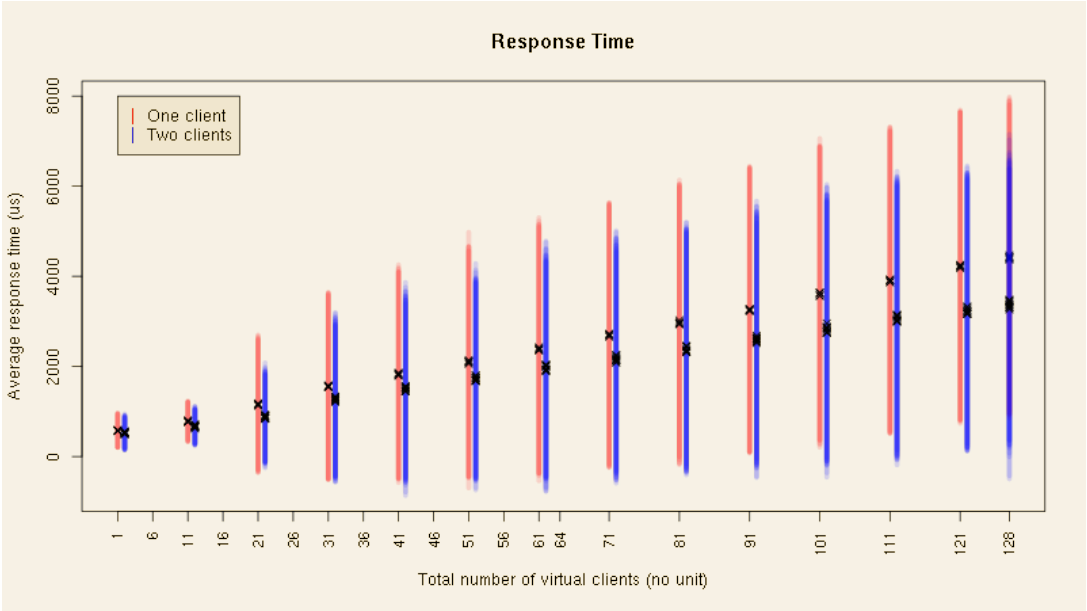

Figure 5: Response time in the baseline experiment

Number of servers	3
Number of client machines	3
Virtual clients / machine	64 (explain if chosen otherwise)
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Replicate to all (R=3)
Runtime x repetitions	1h x 1
Log files	stability-client0, stability-client1, stability-client2, stability-middle

Figure 6: Stability trace experiment details

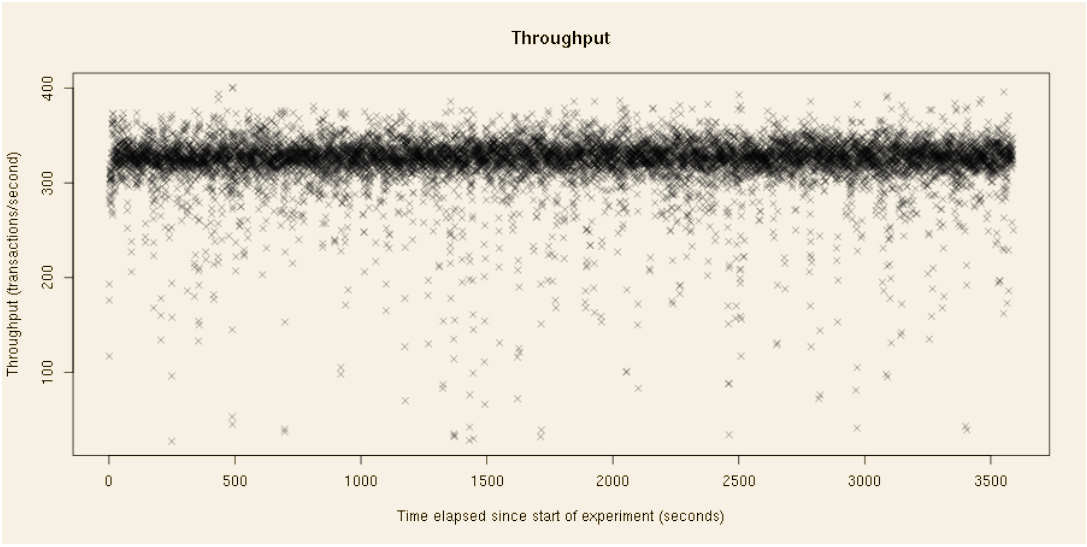

Figure 7: Throughput in the stability trace experiment

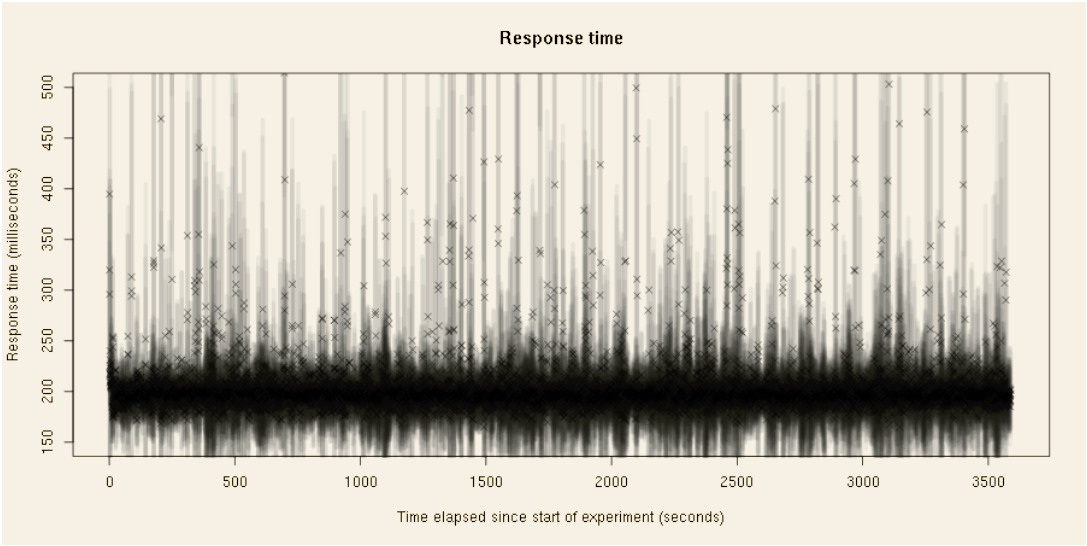

Figure 8: Response time in the stability trace experiment

slow. It only handles about 330 transactions per client per second.

3.3 Response time

In figure 8, the average response time for each client is plotted in the form of crosses. On top of that, wide, semi-transparent error bars are plotted that display the standard deviation of each average. This means that the graph gives a rough idea of the distribution of the actual response times. The graph clearly shows that the response time is very stable around 200 milliseconds with occasional outliers.
Note that these numbers for throughput and response time make sense according to the interactive response time law.

3.4 Overhead of middleware

Even though the middleware is very stable, it is also very expensive. Looking at the above graphs, using the middleware is 100 times more expensive than not using the middleware in

	Average throughput (T/s)	Average Response time (us)
Without middleware	35000	2500
With middleware	330	200000
Overhead Factor	1%	80x

Figure 9: Overhead

terms of throughput. In terms of response time, using the middleware is 80 times more expensive than not using the middleware. These numbers are listed in greater detail in figure 9

Logfile listing

Short name	Location
baseline	https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/results/remote-baseline-experiment-results.csv
stability-client0	https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/results/remote-stability-trace/remote-stability-trace-stabilitytmpresults-0
stability-client1	https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/results/remote-stability-trace/remote-stability-trace-stabilitytmpresults-1
stability-client2	https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/results/remote-stability-trace/remote-stability-trace-stabilitytmpresults-2
stability-middle	https://gitlab.inf.ethz.ch/tomk/asl-fall16-project/blob/master/results/remote-stability-trace-results.csv

References

[1] Arrays (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html.

[2] Blockingqueue (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html.