Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Your name***
Legi number: ***Your legi number***

Grading

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

Notes on writing the report **(remove this page for submission)**

The report for first milestone not need to be extensive but it must be concise, complete, and correct. Conciseness is important in terms of content and explanations, focusing on what has been done and explanations of the results. A long report is not necessarily a better report, especially if there are aspects of the design or the experiments that remain unexplained. Completeness implies that the report should give a comprehensive idea of what has been done by mentioning all key aspects of the design, experiments, and analysis. Aspects of the system, be it of its design or of its behavior, that remain unexplained detract from the credibility of the report. Correctness is expected in terms of the explanations being logical and correlate with the numbers in the experiments and the design.

Remember that this is a report about the system you have designed and built, about the experiments you have performed, and about how you interpret the results of the experiments and map them to your design and implementation. Please do not contact us seeking confirmation and assurances about, e.g., whether the report is sufficient, your interpretation of the data, validation of concrete aspects of your design, or whether you have done enough experiments. Making those decisions is your job and part of what the course will evaluate.

The report will be graded together with the code and data submitted. **The maximum number of points is 200 for each of the milestones and you will need at least 100 to pass. Keep in mind that to pass the project you need to collect at least 400 points from the three milestones.** You might be called for a meeting in person to clarify aspects of the report or the system and to make a short presentation of the work done. By submitting the report, the code, and the data, you confirm that you have done the work on your own, the code has been developed by yourself, the data submitted comes from experiments your have done, you have written the report on your own, and you have not copied neither code nor text nor data from other sources.

A passing grade for the milestone requires at the very minimum:

- Conforming to this template
- A working system
- Consistent experimental results of the entire system
- Internal measurements of the middleware
- Solid and credible explanations of the design, experimental results and behavior of the implemented system

Formatting guidelines

We expect you to use this template for the report, but in case you want to use Word or an other text processor, keep in mind the following:

- We expect you to submit **a single PDF that has the same section structure as this template, and answers all points we outline here**. If you use this file, you should remove this page with notes, and the short description provided by us at the beginning of sections.
- Keep the same cover page as on this document and **fill out your name and legi number**. Leave the grading table empty.
- The main text should be in **single-column format with 11pt font on A4 paper**. In case you don't start with one of the files provided by us, **for margins use 2.54 cm (1 inch) on all sides**.

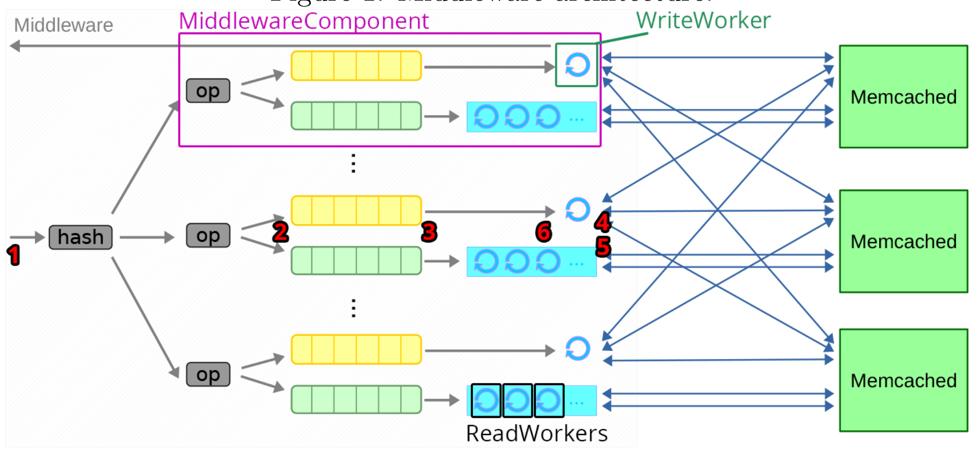
1 System Description

1.1 Overall Architecture

The most important classes in this implementation all belong to the package main.java.asl and are as follows. Note that whenever a Java class is referenced, it will appear as a link to GitLab, e.g. MiddlewareMain.

- MiddlewareMain is responsible for setting up all parts of the middleware.
- Request is a wrapper class for all GET- or SET-requests.
- LoadBalancer is the front of the middleware. It reads all incoming requests from all clients using java.nio, hashes the requests using a Hasher and forwards them to the correct servers for writing or reading.
- UniformHasher implements the Hasher interface and is responsible for mapping request keys to memcached servers. More in Section 1.2.
- MiddlewareComponent is a lightweight class that holds the read and write queues for a given server, and starts the threads that process requests from those queues.
- WriteWorker and ReadWorkerimplement the write and read thread for a given MiddlewareComponent. More in Sections 1.3 and 1.4.

Figure 1: Middleware architecture.



The middleware is instrumented at six points that are also shown in Figure 1:

1. $t_{created}$ – when the request is received in LoadBalancer.
2. $t_{enqueued}$ – when the request is added to the queue.
3. $t_{dequeued}$ – when the request is removed from the queue.
4. $t_{forwarded}$ – when the request has been sent to all the servers (one server for GET requests and R servers for SET requests).
5. $t_{received}$ – when responses to the request have been received from all servers.
6. $t_{returned}$ – when the response is returned to the client.

TODO: "Shortly outline the main design decisions?"

1.2 Load Balancing and Hashing

The hashing is implemented by UniformHasher. The hashing scheme for a given key s works as follows:

1. s is hashed into a 32-bit signed integer using Java's native String.hashCode().
2. The hash is used to set the seed of a random number generator (java.util.Random).
3. The first random number that the generator returns is used.

The uniformity of hashing was also validated in tests (see UniformHasherTest). For 1 million random strings and 13 target machines, the distribution to different machines was the following:

```
Machine   0 got     76706 hits.
Machine   1 got     76896 hits.
Machine   2 got     76718 hits.
Machine   3 got     76829 hits.
Machine   4 got     77102 hits.
Machine   5 got     76980 hits.
Machine   6 got     76467 hits.
Machine   7 got     76940 hits.
Machine   8 got     77194 hits.
Machine   9 got     76896 hits.
Machine  10 got     77107 hits.
Machine  11 got     76785 hits.
Machine  12 got     77380 hits.
```

As apparent, the distribution is indeed uniform.

Selection of replicated machines for a given replication factor R was done by first selecting the primary machine using the scheme described above, and then selecting the next $R-1$ machines for replication. E.g. for a setup with 8 memcached servers and $R=5$, a key whose primary machine is 5 would be replicated to machines 6, 7, 0, and 1.

1.3 Write Operations and Replication

The write operations are handled by WriteWorkers. Each WriteWorker runs on one thread and has exactly one connection to each memcached server it needs to write to, so in total R connections (where R is the replication factor).

WriteWorker runs an infinite while-loop in which it does two distinct things.

Firstly, if there are any requests available in the queue of SET-requests, it removes one request r from the queue. It then writes to each of the replication servers without waiting for a response, i.e. it writes the whole SET-request to the first server, then to the second server, and so on. For the non-replicated case, only one request is sent.

Secondly, WriteWorker checks all memcached servers to see if any of them have responded. This is done in a non-blocking manner using java.nio: if a server is not yet ready to respond, other servers will be checked. WriteWorker keeps track of all responses to the same request and once all servers have returned a response, the worst out of the R responses is forwarded to the client. For the non-replicated case, this process reduces to just forwarding the response from memcached to the client.

TODO: Estimate of latencies – replicated and non-replicated Give an estimate of the latencies the writing operation will incur, and generalize it to the replicated case. What do you expect will limit the rate at which writes can be carried out in the system (if anything)?

1.4 Read Operations and Thread Pool

The read operations are handled by ReadWorkers. Each ReadWorker runs on one thread and has exactly one socket connection to its assigned memcached server.

Every ReadWorker runs an infinite while-loop in which it takes a request r from its assigned queue of GET-requests, writes the contents of r to its assigned memcached server, blocks until the response from memcached arrives, sets the response buffer of r to what it received from memcached and finally sends the response to the client corresponding to r.

Since multiple ReadWorkers read from the queue of GET-requests at the same time as the LoadBalancer is inserting elements, the queue needs to be safe to concurrent access by multiple threads. For this reason, BlockingQueue was chosen; in particular, the ArrayBlockingQueue subclass of BlockingQueue. The maximum size of the queue was set to a constant 200 (defined in MiddlewareMain.QUEUE_SIZE), because 3 load generating machines each with 64 concurrent clients can generate a maximum of $3 \cdot 64 = 192 < 200$ requests at a time, which in the worst (although unlikely) case will all be forwarded to the same server.

2 Memcached Baselines

TODO:

This section will report experimental results. All such parts will start with a short description of the experimental setup. The log files should be identified by a short name, or number, which will be explicitly listed at the end of the document (see Logfile Listing at the end). **If this table is missing or the logfiles listed can't be found in your repository the experiment could be considered invalid, and no points will be awarded!** For baseline measurement of memcached provide **two** graphs (Section 2.1 and 2.2), one with aggregated throughput and one with average response time and standard deviation as a function of number of virtual clients. Increase these in steps from 1 to 128. Give a short explanation of memcache's behavior and find the number of virtual clients that saturate the server.

Number of servers	1
Number of client machines	1 to 2
Virtual clients / machine	1 to 64
Workload	Key 16B, Value 128B, Writes 1% [1]
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	microbench1, microbench2, ...

2.1 Throughput

See previous explanation.

2.2 Response time

See previous explanation.

3 Stability Trace

TODO:

In this section you will have to show that the middleware is functional and it can handle a long-running workload without crashing or degrading in performance. For this you will run it with full replication for one hour connected to three memcache instances and three load generator machines. You

[1] 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.

will have to provide two graphs. The x-axis is time and the y-axis is either throughput or response time. Include standard deviation whenever applicable.

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	trace1, ...

3.1 Throughput

See previous explanation.

3.2 Response time

See previous explanation.

3.3 Overhead of middleware

Compare the performance you expect based on the baselines and the one you observe in the trace and quantify the overheads introduced by the middleware (if any), Look at both response time and achievable throughput when making the comparison. Provide an overview of the overheads in a table form.

Logfile listing

Short name	**Location**
microbench1	`https://gitlab.inf.ethz.ch/.../baseline/logfile.log`
microbench2	`https://gitlab.inf.ethz.ch/.../baseline/logfile2.log`
trace1	`https://gitlab.inf.ethz.ch/.../baseline/logfile.log`
...	...