Advanced Systems Lab Report

Autumn Semester 2017

Name: Simon Haefeli
Legi: 13-918-107

Grading

Section	**Points**
1	
2	
3	
4	
5	
6	
7	
Total	

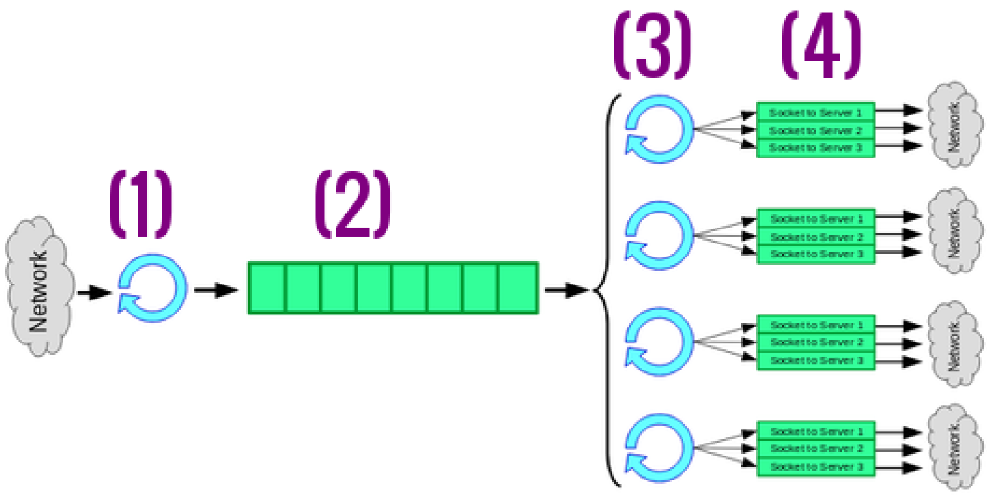
1 System Overview

1.1 Abstract

This projects consisted of designing, implementing and evaluating the performance of a key-value store network system. We added a so called middleware between the memtier benchmark clients and the memcached servers that was able to forward the requests and answers and do some processing for certain particular requests. In this report, we are first discussing the way the middleware has been implemented, analyze results of different experiments and finally modelling our system.

1.2 General design and implementation particularities

Even though the general structure of the middleware was clearly defined, the components of the middlware asked for some important design choices. Here follow a small overview of the different implementation choices.



(1) The net thread is the thread that accepts connections from the clients, and reads the incoming requests to put them in the request queue (2). The *Java.nio* (non-blocking IO) package was used to achieve this purpose: A *ServerSocketChannel* was used as a welcome socket to accept connections from the clients by creating for each client a *SocketChannel*, and a *Selector* was used to iterate over the the sockets (the connection sockets and the welcome socket) to find entering requests for the middleware or connection requests. Using the *Java.nio* allows us not only to abstract the handling of a various number of connections, but also iterates over the client connections in an optimal way by using *SelectionKeys*.

(2) The request queue is a java *LinkedBlockingQueue*, which has the nice properties to be unbounded and beeing FCFS. Requests wait in this queue until a worker thread is free.

(3) The request queue and the worker threads are both managed by the java *ThreadPoolExecutor*. The thread pool executor,which definition is:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long
    keepAliveTime, TimeUnit unit, LinkedBlockingQueue workQueue)
```

has a fixed number of worker threads by setting corePoolSize = maximumPoolSize. Putting a request into the queue or executing it, depending on the availability of the worker threads is beeing done by the same following instruction:

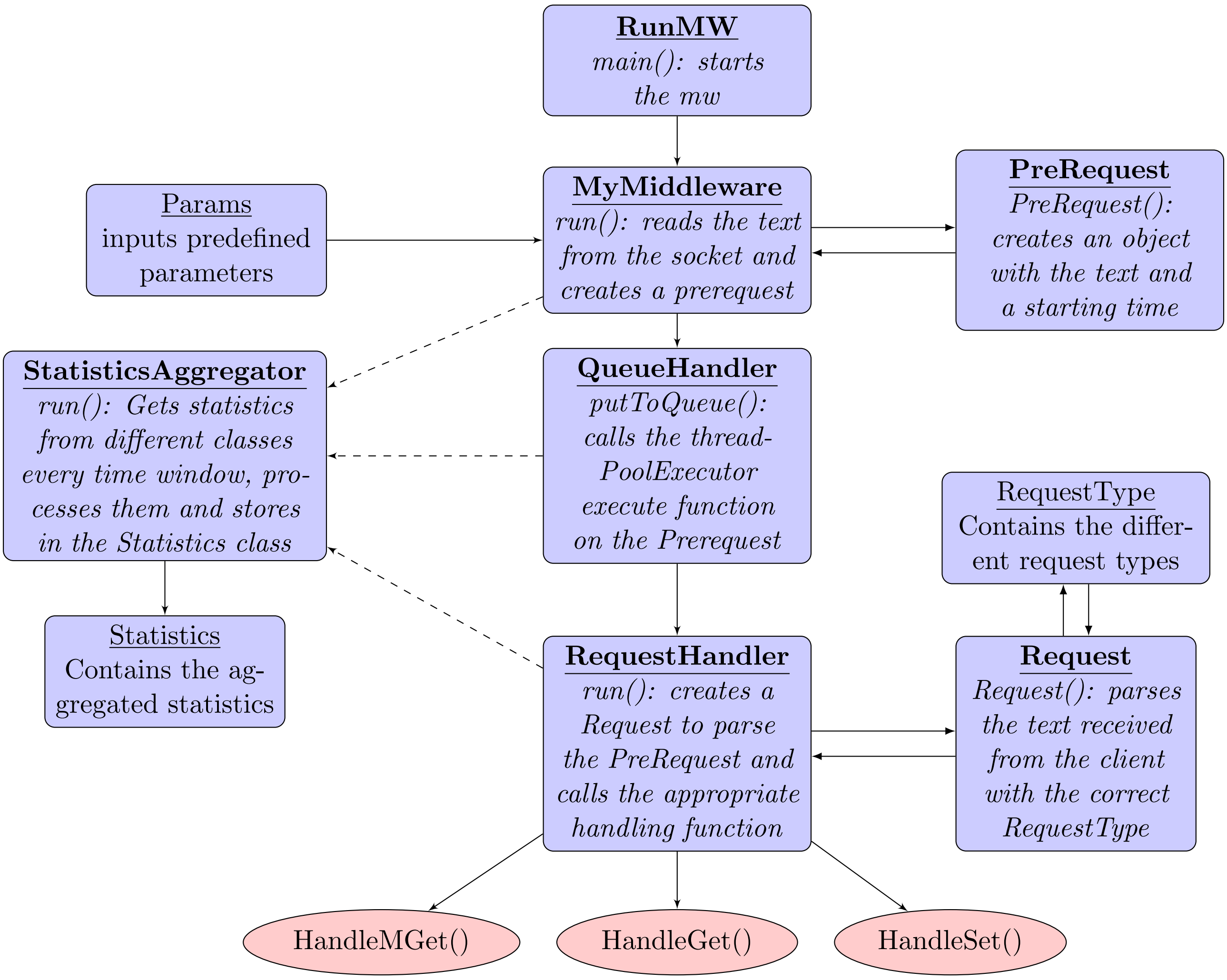
```
myThreadPoolExecutor.execute(Runnable command)
```

The *ThreadPoolExecutor* will execute the command if a worker thread is available, or put it to the unbounded queue if not, which is exactly the expected behaviour.[1]

(4) The connection sockets with the memcached servers are normal Java *Socket*. Each worker thread opens a socket to each memcached server the first time it is beeing run and keeps it open until the middleware is stopped.

1.3 Request handling

We now want to look closer at the implementation of the middleware and how the pipeline looks like for a request beeing processed. The following flow chart illustrates this:



[1]https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html, part on "Queuing"

To make some correpondencies with the general structure of the middleware presented in section 1.2, we see that the *MyMiddleware run()* function represents the net thread, the *Queue-Handler* class has a *ThreadPoolExecutor* field that manages the request queue and the call of the worker thread, and that the *RequestHandler run()* function represents one worker thread. We may also note that a PreRequest class is beeing created before putting the received message into the queue. This is to bind the message with a time, to be able to track the time a message spends in the queue.

At the end of the pipeline, the different requests are treated according to their type (get, set, or multi-get). In case of a set, the request is sent to all memcached servers sequentially, after what the worker thread waits for all the answers, also sequentially, as showed in the next line in the case of 3 memcached servers.

```
send to server 1
send to server 2
send to server 3
wait for answer 1 until received
wait for answer 2 until received
wait for answer 3 until received
merge the answers and send back to client
```

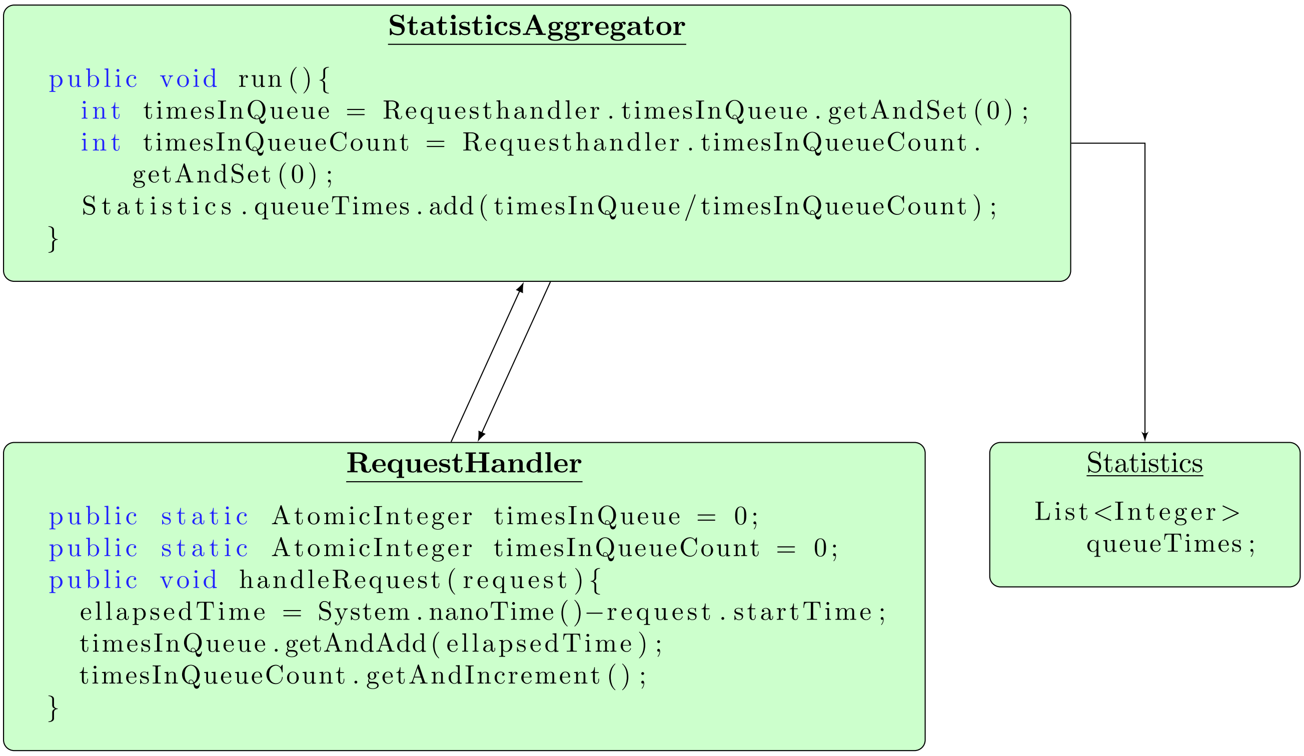
The main drawback of this design is that if one of the server has a higher latency than the other ones, the worker thread might be waiting idle instead of starting to read the answers from the other servers.
In case of a get request, we simply forward the request to a given server. To decide which memcached server we send the request, we simply keep a static field in the *RequestHandler* class which is shared accross the worker threads that keeps track of the last server to which we sent a get request. This variable is then beeing updated in a round-robin way.
In case of a multi-get, two cases are possible: In the non-sharded case, the request is simply beeing executed as a normal get. In the sharded case, the multi get is beeing splitted equally accross the different available memcached servers.

Finally, since we want the connections between the middleware and the servers to be open before running the experiments, we need to initialize these once for every worker thread. By defining the sockets in a *ThreadLocal* in the *RequestHandler* class, the initialization of the sockets will happen only the first time a worker thread is beeing called. That's why every thread is called once during the initialization phase of the middleware, by executing number of worker threads "init" requests, which are fake requests that don't need any special handling.[2]

1.4 Statistics gathering

Statistics have to be made in different classes accross the middleware, either a count (number of gets, number of sets,...), or an average (queue length, time in queue,...). These statistics are beeing stored in static atomic fields in the respective classes, that can be gotten from the *StatisticsAggregator* class, which is a *Runnable* called every second, the first time at initialization of the middleware. These statistics are then collected every given time window, averaged and stored in the *Statistics* class. This process is beeing illustrated in the next figure:

[2] According to the javadoc of the ThreadPoolExecutor class, "When a new task is submitted in method execute(java.lang.Runnable), and fewer than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle". And that's why it is enough to run corePoolSize number of requests to have them run all once and thus initialized their sockets.



The collected statistics in the *Statistics* class are then beeing filtered, by removing the warm-up time (according to some experiments I made, the middlware needs around 10 seconds to get stable data) and the cool down time (1 second), and removing the zero values that are present before and after the significative values (zeros that occur because of the late start of a memtier client for example or an early stop).
Finally, when the middleware is shut down, the average and standart deviation of every statistic is beeing computed and printed to a file. Depending on the experiment, we sometimes also print all the values to construct histograms.

2 Baseline without Middleware (75 pts)

In this experiments you study the performance characteristics of the memtier clients and memcached servers.

2.1 One Server

Both, for a read-only and write-only workload plot the throughput and the response time as a function of NumClients. All clients are connected to a single memcached instance.

Use 3 load generating VMs, with one memtier (CT=2) each, and vary the number of virtual clients (VC) per memtier thread between 1 and 32. Show how the behavior of the server changes as we add more clients.

Number of servers	1
Number of client machines	3
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 or more

2.1.1 Explanation

Describe in which phase the memcached servers are under-saturated, saturated, or over-saturated. Describe how throughput and response time correlate. Explain what further conclusions can be drawn from the experiment.

2.2 Two Servers

For a read-only and write-only workload plot throughput and response time as a function of NumClients. The clients are connected to two memcached instances.

Use 1 load generating VM, with one memtier (CT=1) connected to each memcached instance (two memcache instances in total), and vary the number of virtual clients (VC) per memtier thread between 1 and 32. Show how the behavior of the server changes and explain what conclusions we can draw from this experiment.

Number of servers	2
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	N/A
Worker threads per middleware	N/A
Repetitions	3 or more (at least 1 minute each)

2.2.1 Explanation

Describe how this experiment compares to the previous section. Which results are the same and which ones differ? Explain what further conclusions can be drawn from the experiment.

2.3 Summary

Based on the experiments above, fill out the following table:

Maximum throughput of different VMs.

	Read-only workload	Write-only workload	Configuration gives max. throughput
One memcached server			
One load generating VM			

Write at least two paragraphs about how both results relate. Describe what is the bottleneck of this setup is. If the maximum throughput for both experiments is the same, explain why. If it is not the case, explain why not. Write down key take-away messages about the behaviour of the memtier clients and the memcached servers.

3 Baseline with Middleware (90 pts)

In this set of experiments, you will have to use 1 load generator VM and 1 memcached server, measuring how the throughput of the system changes when increasing the number of clients. Scaling virtual clients inside memtier has to be done as explained in the previous sections. Plot both throughput and response time as measured on the middleware.

3.1 One Middleware

Connect one load generator machine (one instance of memtier with CT=2) to a single middleware and use 1 memcached server. Run a read-only and a write-only workload with increasing number of clients (between 2 and 64) and measure response time *both at the client and at the middleware*, and plot the throughput and response time measured in the middleware.

Repeat this experiment for different number of worker threads inside the middleware: 8, 16, 32, 64.

Number of servers	1
Number of client machines	1
Instances of memtier per machine	1
Threads per memtier instance	2
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

3.1.1 Explanation

Provide a detailed analysis of the results (e.g., bottleneck analysis, component utilizations, average queue lengths, system saturation). Add any additional figures and experiments that help you illustrate your point and support your claims.

3.2 Two Middlewares

Connect one load generator machine (two instances of memtier with CT=1) to two middlewares and use 1 memcached server. Run a read-only and a write-only workload with increasing number of clients (between 2 and 64) and measure response time *both at the client and at the middleware*, and plot the throughput and response time as measured in the middleware.

Repeat this experiment for different number of worker threads inside the middleware: 8, 16, 32, 64.

If in your experiment the middleware is not the bottleneck, repeat the experiment that reaches the highest throughput but using two load generator VMs (each with 2x memtier CT=1) instead of one. Otherwise, explain how you know that the middlewares are the limiting factor in terms of throughput.

Number of servers	1
Number of client machines	1
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..32]
Workload	Write-only and Read-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

3.2.1 Explanation

Provide a detailed analysis of the results (e.g., bottleneck analysis, component utilizations, average queue lengths, system saturation). Add any additional figures and experiments that help you illustrate your point and support your claims.

3.3 Summary

Based on the experiments above, fill out the following table. For both of them use the numbers from a single experiment to fill out all lines. Miss rate represents the percentage of GET requests that return no data. Time in the queue refers to the time spent in the queue between the net-thread and the worker threads.

Maximum throughput for one middleware.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware				
Reads: Measured on clients			n/a	
Writes: Measured on middleware				n/a
Writes: Measured on clients			n/a	n/a

Maximum throughput for two middlewares.

	Throughput	Response time	Average time in queue	Miss rate
Reads: Measured on middleware				
Reads: Measured on clients			n/a	
Writes: Measured on middleware				n/a
Writes: Measured on clients			n/a	n/a

Based on the data provided in these tables, write at least two paragraphs summarizing your findings about the performance of the middleware in the baseline experiments.

4 Throughput for Writes (90 pts)

4.1 Full System

Connect three load generating VMs to two middlewares and three memchached servers. Run a write-only experiment. You need to plot throughput and response time measured on the middleware as a function of number of clients. The measurements have to be performed for 8, 16, 32 and 64 worker threads inside each middleware.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	[1..32]
Workload	Write-only
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	2
Worker threads per middleware	[8..64]
Repetitions	3 or more (at least 1 minute each)

4.1.1 Explanation

Provide a detailed analysis of the results (e.g., bottleneck analysis, component utilizations, average queue lengths, system saturation). Add any additional figures and experiments that help you illustrate your point and support your claims.

4.2 Summary

Based on the experiments above, fill out the following table with the data corresponding to the maximum throughput point for all four worker-thread scenarios.

Maximum throughput for the full system

	WT=8	WT=16	WT=32	WT=64
Throughput (Middleware)				
Throughput (Derived from MW response time)				
Throughput (Client)				
Average time in queue				
Average length of queue				
Average time waiting for memcached				

Based on the data provided in these tables, draw conclusions on the state of your system for a variable number of worker threads.

5 Gets and Multi-gets (90 pts)

For this set of experiments you will use three load generating machines, two middlewares and three memcached servers. Each memtier instance should have 2 virtual clients in total and the number of middleware worker threads is 64, or the one that provides the highest throughput in your system (whichever number of threads is smaller).

For multi-GET workloads, memtier will generate a mixture of SETs, GETs, and multi-GETs. Memtier only allows to specify the maximum number of keys in a multi-GET request. Therefore, be aware that requests can also contain fewer keys than the provided value. It is recommended to record the average size of the multi-GETs. You will have to measure response time on the client as a function of multi-get size, with and without sharding on the middlewares.

5.1 Sharded Case

Run multi-gets with 1, 3, 6 and 9 keys (memtier configuration) with sharding enabled (multi-gets are broken up into smaller multi-gets and spread across servers). Plot average response time as measured on the client, as well as the 25th, 50th, 75th, 90th and 99th percentiles.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	memtier-default
Multi-Get behavior	Sharded
Multi-Get size	[1..9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.
Repetitions	3 or more (at least 1 minute each)

5.1.1 Explanation

Provide a detailed analysis of the results (e.g., bottleneck analysis, component utilizations, average queue lengths, system saturation). Add any additional figures and experiments that help you illustrate your point and support your claims.

5.2 Non-sharded Case

Run multi-gets with 1, 3, 6 and 9 keys (memtier configuration) with sharding disabled. Plot average response time as measured on the client, as well as the 25th, 50th, 75th, 90th and 99th percentiles.

Number of servers	3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	2
Workload	memtier-default
Multi-Get behavior	Non-Sharded
Multi-Get size	[1..9]
Number of middlewares	2
Worker threads per middleware	max. throughput config.
Repetitions	3 or more (at least 1 minute each)

5.2.1 Explanation

Provide a detailed analysis of the results (e.g., bottleneck analysis, component utilizations, average queue lengths, system saturation). Add any additional figures and experiments that help you illustrate your point and support your claims.

5.3 Histogram

For the case with 6 keys inside the multi-get, display four histograms representing the sharded and non-sharded response time distribution, both as measured on the client, and inside the middleware. Choose the bucket size in the same way for all four, and such that there are at least 10 buckets on each of the graphs.

5.4 Summary

Provide a detailed comparison of the sharded and non-shareded modes. For which multi-GET size is sharding the preferred option? Provide a detailed analysis of your system. Add any additional figures and experiments that help you illustrate your point and support your claims.

6 2K Analysis (90 pts)

For 3 client machines (with 64 total virtual clients per client VM) measure the throughput and response time of your system in a 2k experiment with repetitions. All GET operations have a single key. Investigate the following parameters:

- Memcached servers: 2 and 3
- Middlewares: 1 and 2
- Worker threads per MW: 8 and 32

Repeat the experiment for (a) a write-only, (b) a read-only, and (c) a 50-50-read-write workload. For each of the three workloads, what is the impact of these parameters on throughput, respectively response time?

Number of servers	2 and 3
Number of client machines	3
Instances of memtier per machine	2
Threads per memtier instance	1
Virtual clients per thread	32
Workload	Write-only, Read-only, and 50-50-read-write
Multi-Get behavior	N/A
Multi-Get size	N/A
Number of middlewares	1 and 2
Worker threads per middleware	8 and 32
Repetitions	3 or more (at least 1 minute each)

7 Queuing Model (90 pts)

Note that for queuing models it is enough to use the experimental results from the previous sections. It is, however, possible that the numbers you need are not only the ones in the figures we asked for, but also the internal measurements that you have obtained through instrumentation of your middleware.

7.1 M/M/1

Build queuing model based on Section 4 (write-only throughput) for each worker-thread configuration of the middleware. Use one M/M/1 queue to model your entire system. Motivate your choice of input parameters to the model. Explain for which experiments the predictions of the model match and for which they do not.

7.2 M/M/m

Build an M/M/m model based on Section 4, where each middleware worker thread is represented as one service. Motivate your choice of input parameters to the model. Explain for which experiments the predictions of the model match and for which they do not.

7.3 Network of Queues

Based on Section 3, build a network of queues which simulates your system. Motivate the design of your network of queues and relate it wherever possible to a component of your system. Motivate your choice of input parameters for the different queues inside the network. Perform a detailed analysis of the utilization of each component and clearly state what the bottleneck of your system is. Explain for which experiments the predictions of the model match and for which they do not.