Advanced Systems Lab (Fall'16) – First Milestone

Name: ***Eduardo Pedroni***
Legi number: ***16-930-596***

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 middleware design consists of a small number of classes. `Middleware`[1] is responsible for creating and managing all middleware components. Throughout the execution of the middleware, it asynchronously listens for new client connections and handles incoming requests from existing clients. `Request`[2] provides a representation of a client request containing pre-extracted metadata such as the request key and type, as well as the connection to the client whence it originated. `Requests`[3] contains a static pool of `Request` objects instantiated on demand and recycled for the lifetime of the application. This class also commits timing information to the request log when requests are recycled. `ReadRunnable`[4] and `WriteRunnable`[5] handle requests synchronously and asynchronously, respectively, acquired from separate queues for read and write requests. `Memcached`[6] is a representation of a running instance of memcached, containing the server's read and write queues. `MiddlewareLog`[7] provides a pre-configured logger for logging timing information.

Java NIO's `ByteBuffer` offers two options for buffer allocation: direct and non-direct. According to Oracle's Java API specification, "given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it." [1] The middleware therefore makes use of directly-allocated buffers to avoid the overhead of copying data to intermediate buffers when performing I/O operations. Direct buffer allocation is, however, more expensive than non-direct allocation. For this reason, the system was designed to allocate direct buffers early on and reuse them whenever possible.

Since each `Request` object has its directly-allocated buffer, creating a new instance for each incoming client request would be inefficient. To avoid instantiation overhead, `Request` objects are instead acquired via the `Requests.getRequest()` method, which returns recycled requests from its internal pool if any are available. Acquired requests are explicitly returned to the pool by `ReadRunnable` and `WriteRunnable` once they are completed.

The system collects the following timestamps for each request (refer to Figure 1):

- T_{mw_in}: the time when the request is read from the client by the main thread;
- T_{mw_out}: the time when the response to the request is sent back to the client;
- T_{queue_in}: the time when the request is added to its respective queue;
- T_{queue_out}: the time when the request is removed from its respective queue;
- T_{server_in}: the time when the request is written to the memcached server (in the replication scenario, the time when the request is written to the first server);
- T_{server_out}: the time when the response is read back form the memcached server (in the replication scenario, the time when the last response comes in).

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

[2] https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/src/ch/ethz/epedroni/asl/requests/Request.java

[3] https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/src/ch/ethz/epedroni/asl/requests/Requests.java

[4] https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/src/ch/ethz/epedroni/asl/memcached/ReadRunnable.java

[5] https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/src/ch/ethz/epedroni/asl/memcached/WriteRunnable.java

[6] https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/src/ch/ethz/epedroni/asl/memcached/Memcached.java

[7] https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/src/ch/ethz/epedroni/asl/log/MiddlewareLog.java

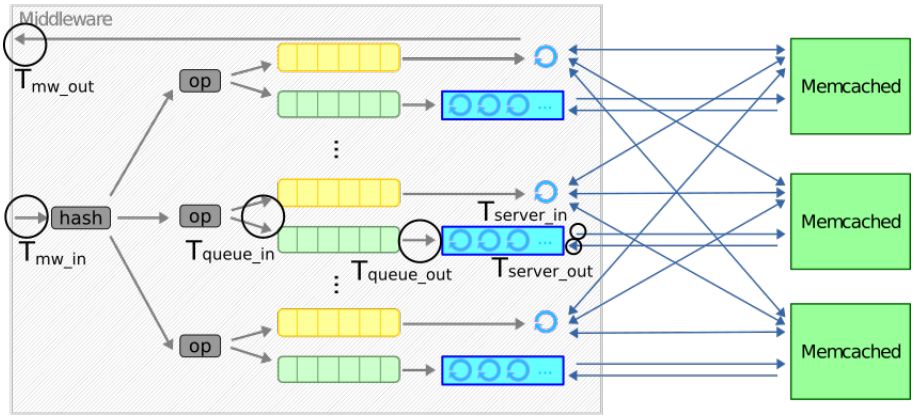

Figure 1: Middleware instrumentation points.

1.2 Load Balancing and Hashing

Load balancing is done by hashing the request key and selecting the primary server using the modulo operation with the number of servers. The absolute result is used as `Arrays.hashCode()` returns a signed integer:

```
private int distributeRequest(Request request) {
    int hash = Arrays.hashCode(request.getKey());
    int server = Math.abs(hash % servers.length);
    return server;
}
```

The standard Java hash function was chosen over more complex hashes for its simplicity and comparatively high performance. Distributing with the modulo operator leads to a reasonable distribution:

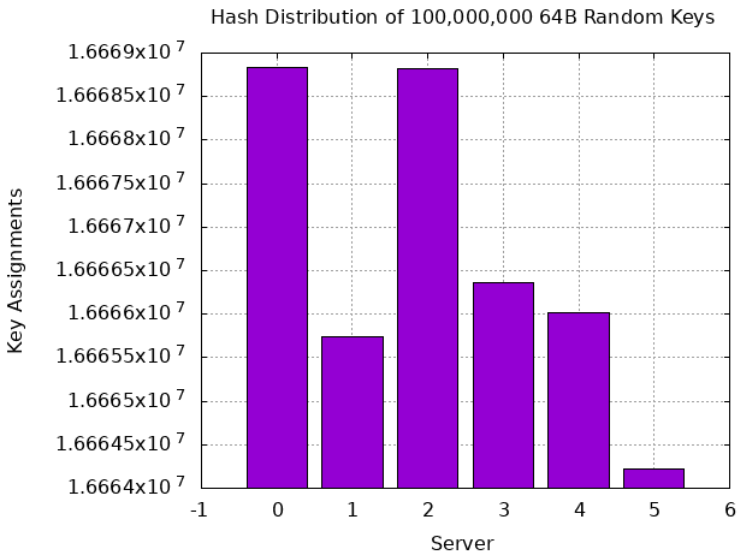

Figure 2: Hash balancing distribution.

The exact test is available in `DistTest.java`[8].

[8] `https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/keydistribution/DistTest.java`

1.3 Write Operations and Replication

Each instance of `WriteRunnable` is given an array of `Memcached` objects upon creation. The first element of the array is the thread's primary server, and subsequent elements are the replication servers. With replication factor 1, the array passed contains a single element, the primary server. The replication sub-arrays are created in `Middleware` by cycling through the main array of servers as the write threads are created. In its constructor, `WriteRunnable` creates `SocketChannel` connections to all provided servers. These connections are configured as non-blocking and registered on the runnable's internal selector.

In the simple, no-replication scenario, the thread attempts to dequeue a request from the queue using the `take()` method, which blocks until it succeeds. When a request is finally acquired, the thread proceeds to select channels. The single available channel is immediately selected (assuming the server is available for writing), the write thread sends out the requests and loops back to selection. The single channel is selected again once the server has responded, and the single response is forwarded to the client as-is.

In the replication scenario, the thread starts off the same by dequeuing a request. Key selection normally yields multiple channels ready for writing, so the request is written out to all servers. Responses are read into a dedicated intermediate receiver buffer as they come in from the servers, and possibly copied to the response buffer depending on their content. The response buffer is only overwritten if:

- it is the first response received for this request; or
- it is negative (NOT_STORED) and the currently buffered response is positive (STORED).

This ensures that clients do not receive false positives if one or more servers are unable to store the key-value pair. Once all responses are received, the next request is acquired from the request queue.

As it is implemented, the system is unable to handle more than one write request at once. In the no-replication scenario this has no impact on performance; however, when replicating requests, some channels might be idle if one of the servers takes long to respond. This implies that the throughput for write requests is fundamentally tied to the performance of the slowest server.

1.4 Read Operations and Thread Pool

Unlike write threads, each instance of `ReadRunnable` is always given a single `Memcached` object upon creation and creates a single connection to it. Connections are set to blocking mode, and no selector is necessary. The system features a direct relation between the number of read threads in the thread pool and the number of connections to servers, since each read thread maintains one connection to its server.

During normal execution, each read thread attempts to dequeue a request from its request queue using the blocking dequeue operation `take()`. When a request is acquired, the thread simply writes it to its server. The response is read back and directly forwarded to the client without further processing, and the thread moves on to the next request.

If more than one read thread per server is employed, the server's read queue will be subject to concurrent accesses by those threads. This is addressed by using a thread-safe queue implementation provided by the Java API: `LinkedBlockingQueue`. Internally, this class uses two reentrant locks, `takeLock` and `putLock`, to ensure that each end of the linked list is protected against simultaneous concurrent accesses. Iterating over the list with an `Iterator` requires both locks to avoid race conditions between calls to `hasNext()` and `next()`. Enqueuing and dequeuing, however, can be performed with only their respective locks; this means that the main middleware thread can add requests to the tail of the queue even as the read threads take from the head, eliminating one potential bottleneck.

2 Memcached Baselines

Memcached baseline experiments were performed using the following parameters:

Number of servers	1
Number of client machines	1 to 2
Virtual clients / machine	1 to 64 in steps of 4
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Not present
Runtime x repetitions	30s x 5
Log files	bl1client*, bl2client*

2.1 Throughput

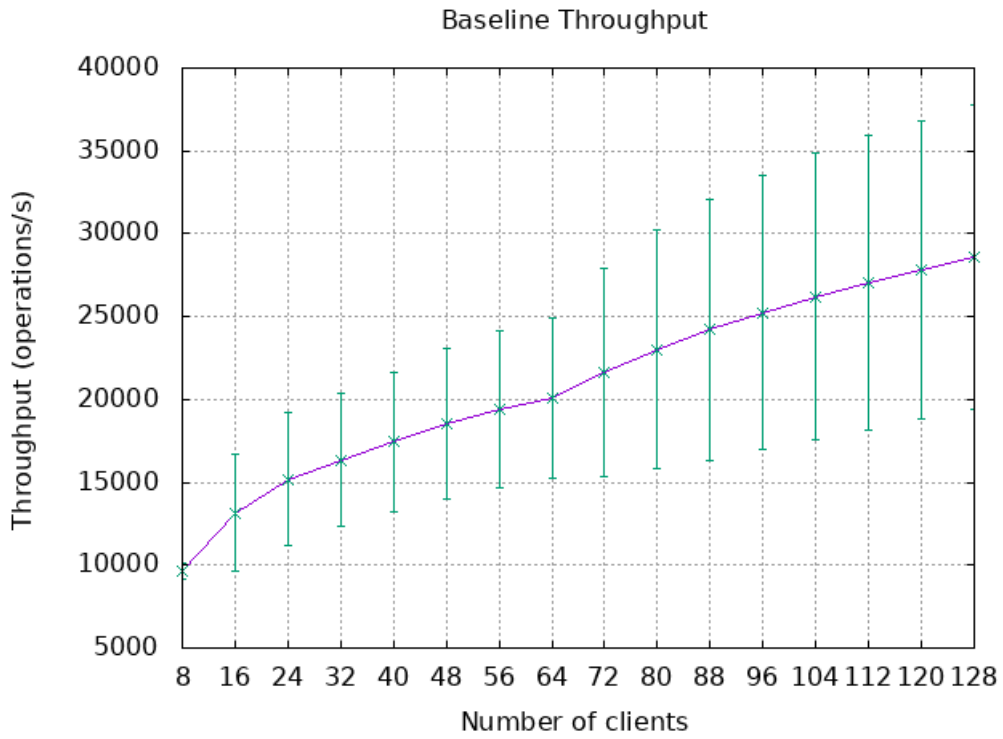

Figure 3: Baseline throughput as a function of number of clients.

Figure 3 shows an increase in throughput as the number of clients rises. This trend is expected, as memcached is capable of serving a small number of clients while still spending time idling. The slight notch at 64 clients is due to the transition from one to two client machines. Even serving requests from 128 clients, memcached shows no definite signs of saturation, though a gradual plateauing tendency is noticeable. The increased standard deviation with greater numbers of clients supports this trend, indicating that the server cannot reliably maintain performance at higher loads. Further experiments with more clients would likely show a throughput peak followed by a sharp decline as the server becomes saturated and client requests begin to build up.

2.2 Response time

Figure 4 shows a steady increase in response time as the number of clients increases. Once again, the notch at 64 clients is due to the transition from one to two client machines. The behaviour displayed is expected for one instance of memcached running on a single thread. As more clients connect to memcached and send requests, any time spent addressing one request is time during which all other requests must wait. The sharp increase in standard deviation further illustrates this, as the response

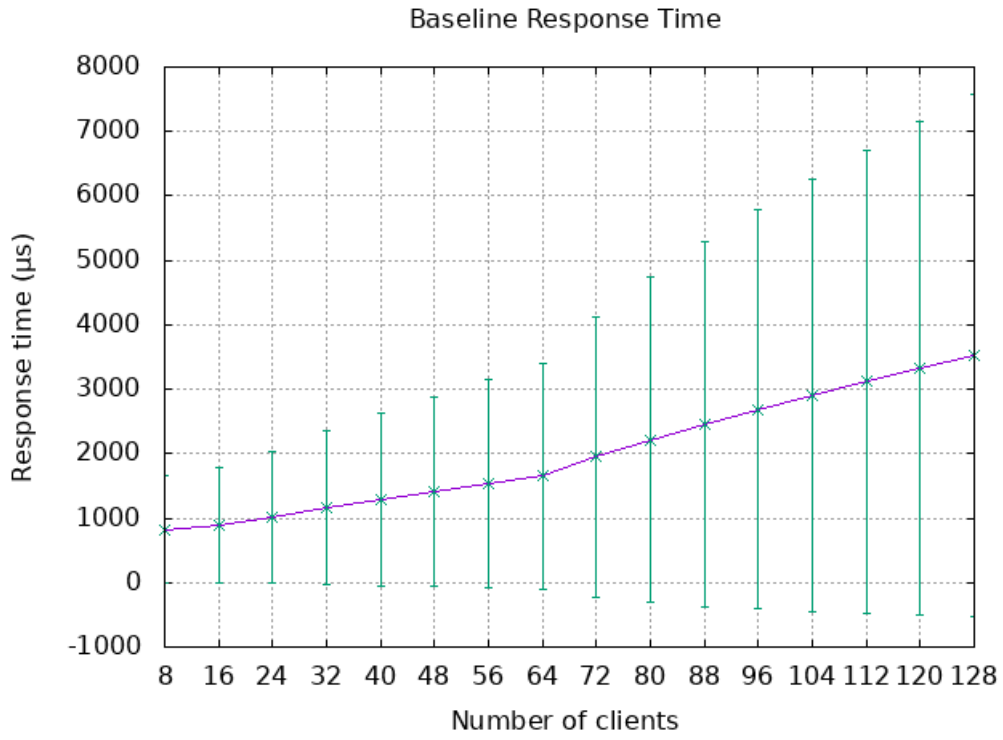

Figure 4: Baseline throughput as a function of number of clients.

time becomes increasingly inconsistent. With many clients connected, the mean response time ceases to be relevant. Due to such high standard deviation, the median response or a percentile plot would paint a clearer picture of the server's actual behaviour.

3 Stability Trace

The middleware was tested for stability and load response with the following parameters:

Number of servers	3
Number of client machines	3
Virtual clients / machine	64
Workload	Key 16B, Value 128B, Writes 1%
Middleware	Replicate to all (R=3)
Read threads	20
Runtime x repetitions	1h x 1
Log files	memalog{1,2,3}, mwlog

3.1 Throughput

Figure 5 shows an aggregation of the throughput from the three memaslap clients during the one-hour stability trace. The middleware is clearly stable, able to run for one hour showing no signs of performance degradation. The throughput fluctuates considerably between approximately 17,000 and 15,000 operations per second.

3.2 Response time

Figure 6 shows an aggregation of the response times of the three memaslap clients during the one-hour stability trace. The data was calculated by averaging the response times and taking the root mean square of the standard deviation of each logged response time sample. No performance degradation is evident, though the standard deviation is too high to draw meaningful conclusions from this plot.

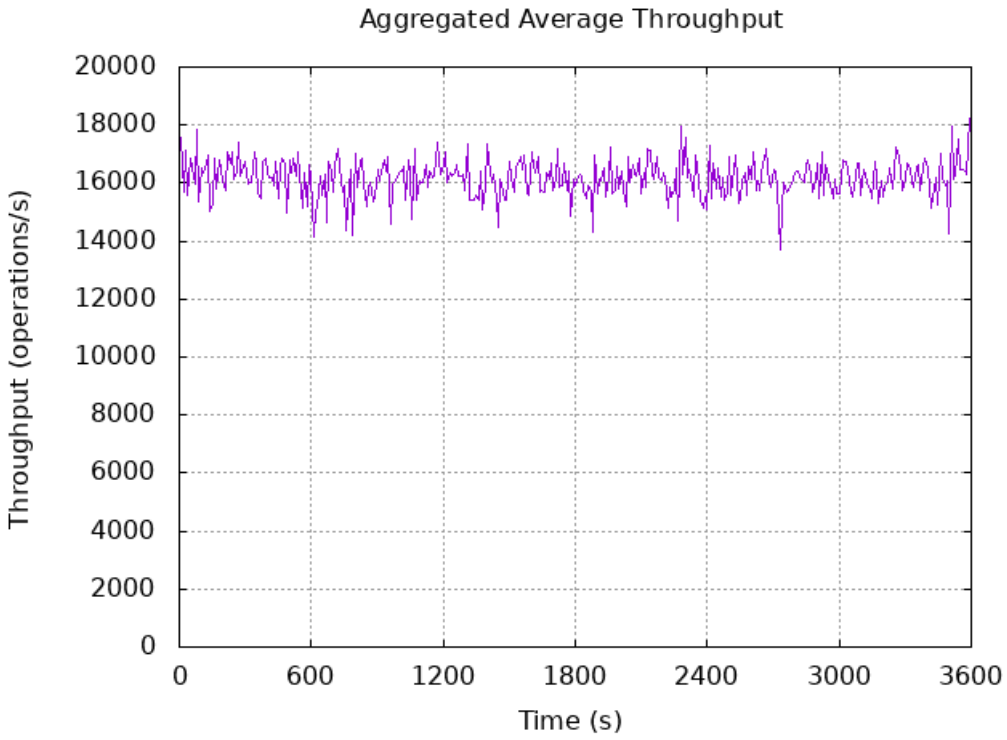

Figure 5: Aggregated middleware throughput as a function of time.

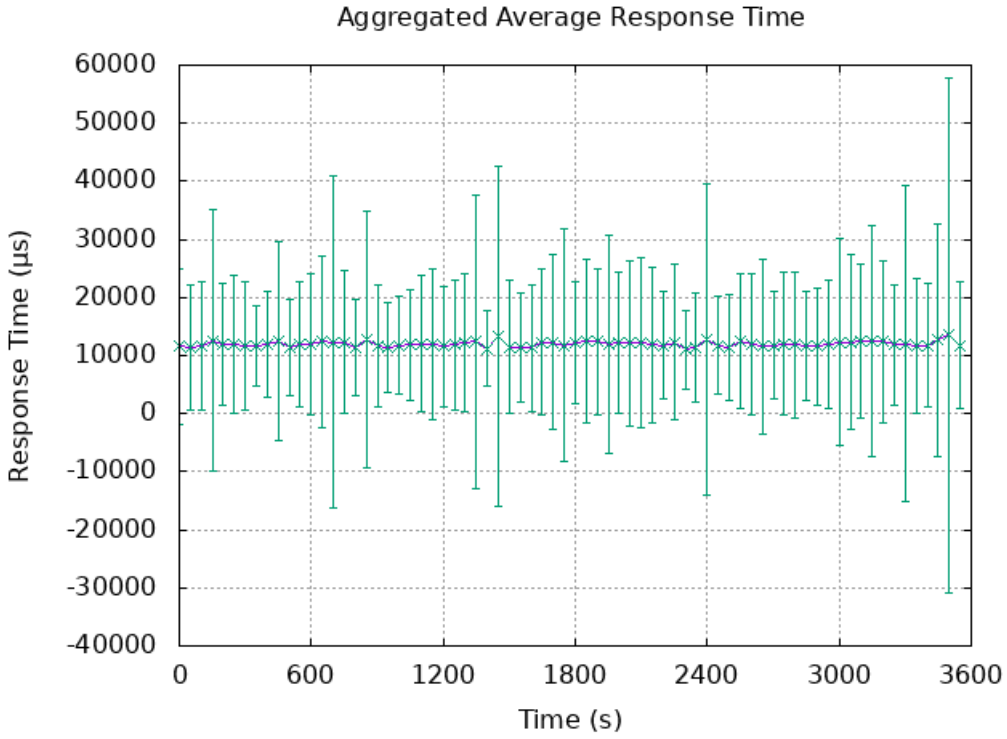

Figure 6: Aggregated middleware response time as a function of time.

3.3 Overhead of Middleware

Memcached's highest average throughput visible in the logged baseline data is approximately 28,000 operations per second, contrasted with 16,000 for the middleware. Memcached performs with a maximum average response time of approximately 3,500 μs against the middleware's average response time of about 12,000 μs. As expected, the middleware introduces considerable overhead, apparent in the throughput and response time measurement differences. The total overhead observed is most likely caused by a combination of the full replication factor and additional network latency associated with the extra hop.

Logfile listing

Short name	Location
bl1client*	https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/baseline/results/1cm/
bl2client*	https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/baseline/results/2cm/
memalog1	https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/trace/mem1.log
memalog2	https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/trace/mem2.log
memalog3	https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/trace/mem3.log
mwlog	https://gitlab.inf.ethz.ch/epedroni/asl-fall16-project/blob/master/reports/m1report/trace/middleware.log

References

[1] *Java Platform, Standard Edition 8 API Specification.* https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html.