Advanced Systems Lab

Autumn Semester 2017

Gustavo Alonso

1 Course Overview

1.1 Prerequisites

This is a project based course operating on a self-study principle. The course relies on the students own initiative, aside from tutorials discussing basic supporting material and exercises covering examples of performance evaluation, there are no lectures in the conventional sense. The project as well as the course requires that the student complements the material discussed in the tutorials and exercises with his/her own study as not all relevant and/or needed material will be covered. The necessary pointers to the textbook and available literature will be provided. The grading will be based solely on the project; there is no exam. The project is to be completed by each student independently.

The pre-requisites for this course include knowledge at the Bachelors level of the following topics:

- Algorithms and data structures
- Programming in Java (General knowledge of object-oriented programming, multi-threading, concurrency, and debugging)
- Networks (Understanding of how computer networks operate, and the basis of performance evaluation in networks)
- Socket Programming (TCP/IP networking)
- System Design (Operating systems, concurrent systems, low level optimizations, and instrumentation)
- Statistics

These topics will not be covered during the course but knowledge of them is assumed. If a student lacks the necessary background, it is expected that he/she will acquire the necessary knowledge on his/her own. No course or project requirements will be changed for individuals because of lack of previous knowledge in any of these areas. Students who do not have the necessary background or system development skills may want to consider taking instead one of the other two Interdisciplinary Labs offered by the department to comply with the requirements for the masters degree in computer science.

1.2 Objectives and General Remarks

The main goal of this course is to learn how to evaluate the performance of complex computer and software systems. The course offers an opportunity to bring together the knowledge and expertise acquired in different areas (networking, systems programming, parallel programming) by allowing students to build a distributed information system.

The course focuses less on system design and development and more on the evaluation and modeling of the system: understanding the behavior, modeling its performance, code instrumentation, bottleneck detection, performance optimizations, as well as analytical and statistical modeling. The ability to explain the behavior of the system plays a bigger role in the grading than the actual building of the system. However, we expect designs and code that have been thought through – major design mistakes and bad coding practices will affect the grade.

This course is a demanding one, not because the task at hand is in itself difficult but because for many students it is the first time they are confronted with designing a system with several degrees of freedom. Skills from several areas of computer science need to be brought to bear into the problem to solve it.

Furthermore, *waiting until close to the deadline to complete the project will not work on this course.* Starting as early as possible is essential, and aiming at completing it well before the deadline is highly recommended so that there is enough time to analyze and understand what has been built. In this course, the focus is on understanding the system and being able to explain what it does and why it behaves the ways it does. This is more important than the development of the system itself. No matter how much effort goes into the development, if the system or the experimental data are ready only shortly before the deadline, there will be no time for developing a thorough understanding of the performance characteristics.

Experience from last editions of the course show that there are a number of typical problems with the project that occur because of poor time management:

- System works but behavior of performance traces cannot be explained (Why response time grows over time? Why throughput peaks for a given number of clients? What are the bottlenecks in the system?)
- System works but the traces are incomplete, experiments have not been repeated a sufficient number of times to achieve statistical significance, lack of proper statistical treatment of the data. etc.
- Inconsistent results in the data collection with different experiments showing contradictory behavior
- Poor modeling or no explanations for the discrepancies between the model and the system
- Unexplained behavior of the system from a performance perspective

In this project, a number of well established professional practices are highly advisable: create a timetable for your project and adhere to it, monitor your progress so that potential delays can be identified and dealt with early enough, use a version control system for development (will be provided), heavily document all your code, plan your experiments and experimental work, use a database to store your experimental data, keep a journal of experiments and experimental parameters, keep track of result files and data to make the analysis simpler later on. Make sure you have backup copies of your code, data, and report.

1.3 Grading

The course will be evaluated through a project. It consists of three parts: (a) a working system, (b) a report of the conducted experiments and the corresponding analysis, and (c) the data collected throughout the experiments. All elements need to be handed in to obtain a passing grade in the project. In the end a minimum of 400 out of 600 points need to be collected to pass the project. Failing the project implies failing the course. Work in the project will be done on an individual basis. Students might be required to present the results in person and explain the system or results in the report. Failure to submit the results on time will result in failing the project. Please note that students still registered for the course on October 15th, 2017 will be considered as having committed to complete the course and handing in a report on the stated deadline. If no report is submitted before the deadline, the grade for the course will be reported as failed (grade of 1).

The deadline to submit the project is: December 18th, 2017 at 17:00 (CET).

The goal of the project is to build a distributed system on a cloud platform and explain its behavior. Developing a system that works but for which no explanation can be produced on why it behaves the way it does is not sufficient to reach a passing grade. Similarly, collecting experimental data or developing a model that does not explain the behavior of the system is not enough to pass the course. A passing grade is reached by having a stable system, clearly structured experiments, the proper explanations for the data and the model, including its potential discrepancies with the implemented system.

1.4 Academic Integrity

For the work on the project a zero tolerance policy will be applied to plagiarism, i.e., plagiarism of any kind will lead to a failing grade in the course and will be reported to the student administration. It is not allowed to use code or scripts from other students or copy code from anywhere (including the internet). It is also not allowed to use experimental data, models, or results from other students. The code handed in by all students will be automatically checked for compliance with the design specifications and for similarities against all other student submissions from this and the previous year.

1.5 Supporting Material

The following text book (available in the library [1]) can be very useful for the course: *The Art of Computer Systems Performance Analysis*, Raj Jain, Wiley Professional Computing, 1991. Of particular relevance are the following chapters:

- Chapters 1, 2, 3 (General introduction and common terminology)
- Chapters 4, 5, 6 (Workloads)
- Chapter 10 (Data presentation)
- Chapters 12, 13, 14 (Probability and statistics)
- Chapters 16, 17, 18, 20, 21, 22 (Experiment design)
- Chapters 30, 31, 32, 33, 36 (Queuing theory)

[1] `http://tiny.cc/artofcomputer`

Communication Please check the web page of the course [2] regularly. Also, make sure that your teaching assistant is informed of the progress you have made. An e-mail address is available for general questions, requests, and feedback regarding the course [3]. Please note that we will neither answer questions specific to your implementation nor pre-grade your project report. We will also not answer questions about whether what you have done so far is enough to reach a passing grade, etc. We are happy to provide examples of what is required, discuss the techniques needed, and show how to avoid common problems but the final project is entirely under the individual responsibility of the student.

2 Project Details

In this year's project you will work with key-value stores. A key-value store is a simple data storage system that allows clients to store arbitrary data under keys of their liking. They are also called NoSQL databases because they are used similarly to how a traditional relational database would be used, but they relax some durability or correctness requirements in exchange of better performance. In this project, you will work with *memcached* [4], a very commonly used main-memory key-value store.

The workloads that you will run on these servers are fully synthetic, generated using *memtier* [5]. That is, the clients to your middleware will also be exclusively instances of the memtier application.

In this project we will only rely on two memcached operations [6]: `GET` to read a value associated with one (`GET`) or more (multi-`GET`) keys, and `SET` operations to insert or update the value belonging to a key. Throughout this project we will use different mixes of these basic operations (e.g. write-only, read-only, 50%-50% read-write, etc.) on thousands of unique key-value pairs.

2.1 Problem Statement

The project consists of the design and development of a middleware platform for key-value stores as well as the evaluation and analysis of its performance, and development of an analytical model of the system, and an analysis of how the model predictions compare to the actual system behavior. The middleware will have to perform data replication to different servers. While proper replication protocols and recovery after failures is necessary for real-world systems, for the purpose of this project, we focus on the mechanism of reading data from respectively writing data to multiple servers.

2.2 Features to Implement

The middleware is to be positioned in front of several memcached instances (servers). Its behavior is as follows:

- In the case of `SET` operations, the middleware forwards the requests to all servers in order to replicate the data, and waits for a response from all of them. For successfully replicating data, your middleware needs to perform the following:

[2] `http://www.systems.ethz.ch/courses/fall2017/asl`

[3] `mailto:sg-asl@lists.inf.ethz.ch`

[4] `https://memcached.org/`

[5] `https://github.com/RedisLabs/memtier_benchmark/`

[6] `https://github.com/memcached/memcached/blob/master/doc/protocol.txt`

1. Recognize a `SET` operation encoded possibly in multiple network packets.
2. The middleware has to forward the complete `SET` request to all memcached servers. It sends it to all memcached servers before checking for an answer.
3. Wait until all memcached servers report successful execution, a send a single success message to the client (in the same format as memcached would).
4. In case of an error on one or multiple servers, relay one of the error messages.

- For `GET` operations with a single key, the middleware acts as a load balancer and forwards each request to only one of the servers. To distribute load equally you can choose to implement either (a) a round-robin load balancer or (b) parse the keys and hash them to determine the index of the server to read from. No load-dependent schemes are required. We do expect, however, some proof that on average all memcached servers are subject to the same load.

- `MULTI-GET`: In the case of `GET` operations with multiple keys, the middleware has to support two modes:

  1. In the first mode (non-sharded read), multi-`GET` operations are treated as regular requests and are forwarded to one server which handles the entire request. In this case, the answer from the server will contain multiple values and will be significantly larger than a single value. Your middleware has to be able to handle up to 10 values in the same response.
  2. In the second mode (sharded read), the middleware splits the multi-`GET` into a set of smaller multi-GET requests (not single GET requests), one request for each server. The middleware should send out all requests before attempting to read answers from the servers. The middleware is responsible for assembling the responses and reordering values, if necessary, from all servers before sending the complete response of the initial multi-`GET` back to the client.

As already mentioned, your middleware has to support any mix of `SET`s, `GET`s and multi-`GET`s, with arbitrary length keys and value sizes smaller or equal to 1024 B in size. Furthermore, the middleware has to support the fact that previously stored items can be evicted from the memcached servers over time, in which case the answer to a GET is an "empty" message (see protocol description [7]). This behavior can happen both with single-key and multi-key GETs.

Figure 1 shows the internal structure of the middleware. It operates as follows:

- The middleware accepts connections and requests from clients on a `TCP` port of your choice, specified when you launch the middleware. There is a single thread (net-thread) that listens for incoming requests on the specified port and enqueues them into the request queue.

- The net-thread puts all incoming requests into the internal request queue from which requests are dequeued by worker-threads residing in a thread pool. The thread pool can be fixed size, but its size has to be a parameter given to the middleware at startup time. A maximum number of 128 threads in this thread pool has to be supported.

- Each worker thread in the thread pool must be able to handle all three types of requests (`SET`, `GET`, multi-`GET`). If the clients send a request that does not belong to one of these types, the worker thread has to record this event and discard data until a newline character is encountered.

[7] `https://github.com/memcached/memcached/blob/master/doc/protocol.txt`

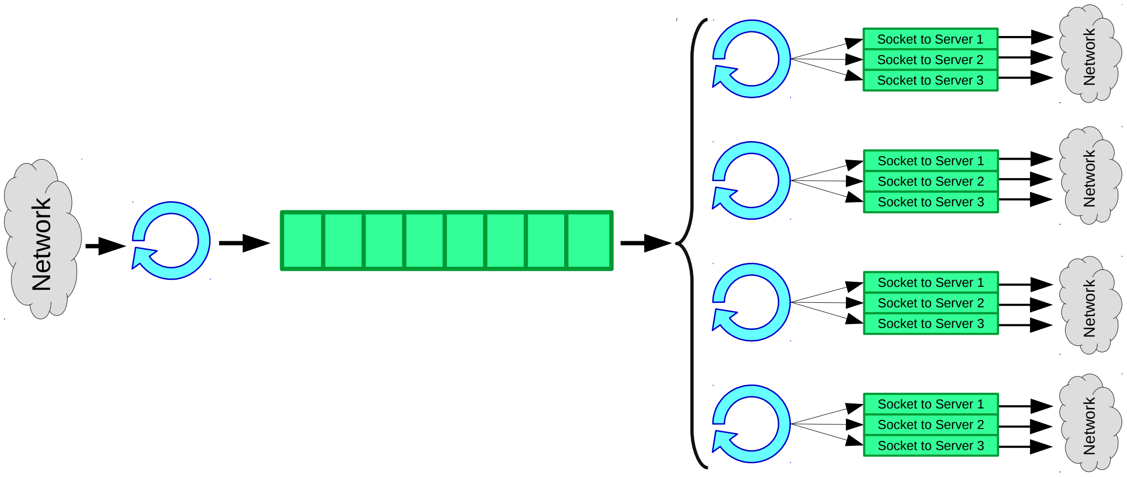

Figure 1: Middleware Architecture depicted with four worker threads and three servers on the back-end. Clients not depicted.

- Each worker thread has a dedicated TCP connection to every memcached server. These connections are set up when the middleware is started and they are only closed when the middleware shuts down.
- Requests are sent by the worker threads to the servers according to the specifications mentioned above. Worker threads wait until an operation has been completed and send the answer from the server to the client before handling the next request.
- At startup time, the middleware uses command line arguments to find the set of servers (at most 3) to connect to, and how many worker threads it should have. The net-thread is always present.
- There should be either a command line interface that allows stopping the middleware, or a hook to catch a "kill" from Linux [8]. The middleware has to print all of its statistics (both final aggregates and smaller resolution ones) when exiting. See more on instrumentation in the next subsection.

2.3 Instrumentation

The system will be benchmarked by generating load with the memtier [9] clients. These clients measure throughput and response time on the client side, but do not provide a detailed break-

[8] http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html

[9] https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached

down of the relative costs inside the middleware and the memcached servers. Therefore, the middleware code needs to be instrumented and report several metrics. Note that due to the multi-threaded architecture of the middleware, statistics should be collected per each thread and merged together when the middleware is stopped.

The middleware has to collect aggregate statistics per, at most, five second windows. These statistics must include:

- Average throughput
- Average queue length
- Average waiting time in the queue
- Average service time of the memcached servers
- Number of `GET`, `SET`, and multi-`GET` operations

At the end of the experiment, the middleware has to output, in addition to the previously described statistics, the following:

- Histogram of the response times in tenth of a millisecond (100μs) steps
- Cache miss ratio (i.e., "empty" responses returned by the memcache servers)
- Any error message or exception that occurred during the experiment

In addition to the metrics mentioned above, we encourage adding instrumentation to any part of the project that is deemed important. Furthermore, we highly recommend using additional statistics and diagnostic tools (like *dstat*) to get additional insight (e.g., CPU utilization, network bandwidth, etc.).

2.4 Middleware Parameters and Deployment in the Cloud

The middleware has to be parameterizable at startup using three parameters:

L: Address to listen on.

P: Port to listen on.

T: Number of memcached servers.

M: IP addresses of the memcached servers.

T: Number of threads in the thread pool.

S: If multi-`GET`s are sharded.

A Java file `RunMW.java` that parses these parameters and starts your middleware will be provided.

All experiments will have to be run on the Microsoft Azure Cloud [10] platform using virtual machines running Linux. Debugging should happen on your local machines, but we will not accept any experiments ran on local computers. The course provides free access to the cloud platform. Please note that the use of such an infrastructure requires discipline and care. The

[10] `https://azure.microsoft.com/en-us/`

course will provide only a finite amount of credit to each student for using the cloud facilities. This amount is more than enough to complete the project.

More details will follow in the exercise sessions and tutorials on the particular steps on getting access to the Azure cloud and receiving vouchers for credit.

The project will require running two middlewares connected to the same set of memcached servers, all running on different virtual machines. Similarly, clients need to connect to the middlewares over the network from different VMs. Each has to be able to connect to at least 3 servers, and accept connections from at least 3 load generating machines. Note that, to avoid fluctuations of performance, each set of experiments that have to be directly comparable must be run on the same set of VMs (that is, all repetitions have to take place without stopping and restarting VMs).

When deploying you system in the Azure cloud, use virtual machines of types *A Basic*, namely A4 (4 cores) for the middleware instances, A2 (2 cores) for the load generators, and A1 (single core) for the memcached servers.

2.5 Report Structure and Deliverables

In order to successfully complete the project, the source code, the report, and the data collected during the experiments have to be handed in. The system has to (a) be stable, (b) be able to forward requests to servers, (c) collect the responses and relay them back to the clients, and (d) report all necessary statistics. The code needs to be documented: each class should have a description of the functionality implemented by that class. The report should contain a section describing how the code implements the design and functionality described in this document. Important implementation decisions affecting the behavior of the system need to be documented.

The detailed structure of the report will be published in the first two weeks of the semester on `https://www.systems.ethz.ch/courses/fall2017/asl`.

3 High-level Overview of Experimental Evaluation

In the following we outline shortly the types of experiments you will have to run in order to complete the report. The more detailed report template will follow shortly.

1. **Clients and Server Baselines**
   Analysis of the behavior and performance of the load generators and the memcached servers. The goal of these experiments is to determine the maximum throughput of (a) the memcached servers and (b) memtier clients for write-only and read-only workloads. These experiments do not involve the middleware.

2. **Middleware Baselines**
   First, the performance of a single middlware needs to investigated. The middleware needs to be connected to one client and one server and investigated under a variety of configurations (e.g., different number of middleware threads). The maximum throughput needs to be determined for each configuration. Next, the experiments need to be performed with a system composed of two middlware servers.

3. **Three Server Configuration**
   Using a write-only workload, two middlwares need to be put in front of three memcached servers in order to determine the effect of replicating data to three servers. The maximum throughput needs to be determined for different middleware configurations (e.g., different number of middleware threads).

4. **Multi-GETs**
   To study the performance of multi-`GET` requests, a system with two middlewares and three servers needs to be evaluated for each of the two modes (i.e., sharded and non-sharded reads). The impact of these two modes needs to be evaluated for different request sizes (i.e., different number of keys contained in the multi-`GET` request).

5. **2k Analysis**
   A 2k analysis needs to be performed for three parameters, namely the number of servers (2 and 3 servers), the number of middleware threads (8 and 32 threads per middleware), and the number of middlwares (1 and 2 middlewares). The impact of these parameters on both the throughput and the response time needs to be calculated and explained.

6. **Queueing Theory**
   The goal of this section is to develop an analytical queuing model for each component of the system and for the system as a whole. Using the model, derive the performance characteristics that the model predicts and compare them with the results obtained in the experiments. The report needs to state where the measured and predicted values match and don't match. We will ask you to model the whole system first as a black box, using a naive model, and then as a more elaborate network of queues. To successfully model the system as a network of queues you will have to understand how requests are handled within the middleware and how long they spend in various processing stages.

Note that for each part, it is not enough to solely perform the experiments in order to reach a passing grade. Each section needs to contain a detailed explanation describing why the behavior is as it has been observed. Furthermore, the report will have to contain a description of the experimental infrastructure, an overview of the parameters for each experiment, and a description of the statistical treatment of the measured data (e.g., all graphs need to contain reasonable error metrics).

Experiment Files All experiments that are used in the report need to be submitted. There is no need to hand in data generated during test or debugging runs. The data needs to be bundled as an archive (zip or tar). Inside the tar, (at least) the following files should be present:

- `client_NN.log` – where the number NN corresponds to each individual memtier process. The logfile contains all output from the clients.
- `mw_NN.log` – NN can be 00 for a single midleware, or 00 and 01 for two. The file contains all output of a middleware.
- `processed.log` – The file that contains the data used to generate a graph/table/etc. and directly computed from the logs previously mentioned.

The root directory needs to contain a `README` file containing an overview of all the different archives and stating which graph/table in the report has been generated based on which data files.

Length of the Experiments and Repetitions All experiments need to be repeated to be statistically significant. By default, every experiment needs to be repeated three times. More iterations should be performed if the measurements are not stable. All measurements need to be made during stable execution, i.e., excluding warm-up and cool-down phases. This stable phase needs to be at least one minute long.

We expect that all experiments can be run (including the repetitions) within 8 to 10 hours, provided that you automated your experiments and do not require constant manual supervision. To get consistent results within one experiment we advise you to run all repetitions and runs in one go and without restarting the VMs. When VMs are restarted in a cloud environment, they can be allocated to different physical machines in the data center leading to different latencies between the VMs which will affect your measurements. More details about this will be provided during the exercises.