Cloud-Based Data Processing
Assignment 4 - Implementing a highly available URL shortener service in the Amazon Cloud

Handout: **01st December 2020**
Due: **07th January 2021 by 6pm**
Discussions: **09th, 16th, 23rd December 2020 and 13th January 2021 in the tutorial**

1 Introduction

In this exercise, we implement a highly available URL shortener. We build the system with three machines and implement some concepts ourselves instead of using the design which builds around Amazon's offerings from the last exercise.

2 The design

Figure 1 visualizes the design. We see three Amazon EC2 instances with attached *Elastic Block Service* instances. The design is a symmetric one, each machine runs the same tasks.

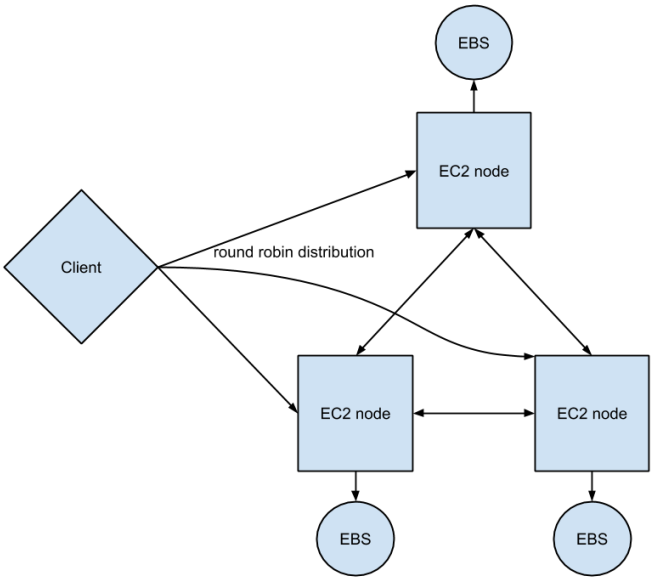

Figure 1: Detailed design of one node

We now detail how every component from the Grokking the System Design Interview model solution is implemented. The *Database Servers* component is implemented by a transactional database running on each Amazon EC2 instance - we recommend using SQLite. The database is synchronously replicated to hold the same data on all three nodes.

We do not implement a *key generation service* but hash the incoming URLs and we use key-space big enough too keep the number of collisions low enough to be handled. The process of handling it is described in subsection 2.2.

The *Cleanup service* is supported by an index on the creation date, maintained by SQLite. Old keys are cleaned up continuously per second.

The *Cache Server* is supported by a simple in-memory mapping structure on each EC2 instance. It is hash-partitioned such that we can choose EC2 instances with less RAM and would be able to scale the cache size by using consistent hashing and adding more nodes in a Dynamo like fashion.

We provide a *Client* which runs as a single, multithreaded program and generates read alias and write alias requests at a ratio of 100:1 and with a key-distribution modelling a typical 80/20 workload. It communicates with your service via a simple REST interface. We describe the client in more detail in subsection 3.1.

Figure 2 shows all components and their interaction on an EC2 instance. The figure also shows the execution of a read and write request which are described in subsection 2.1 and 2.2.

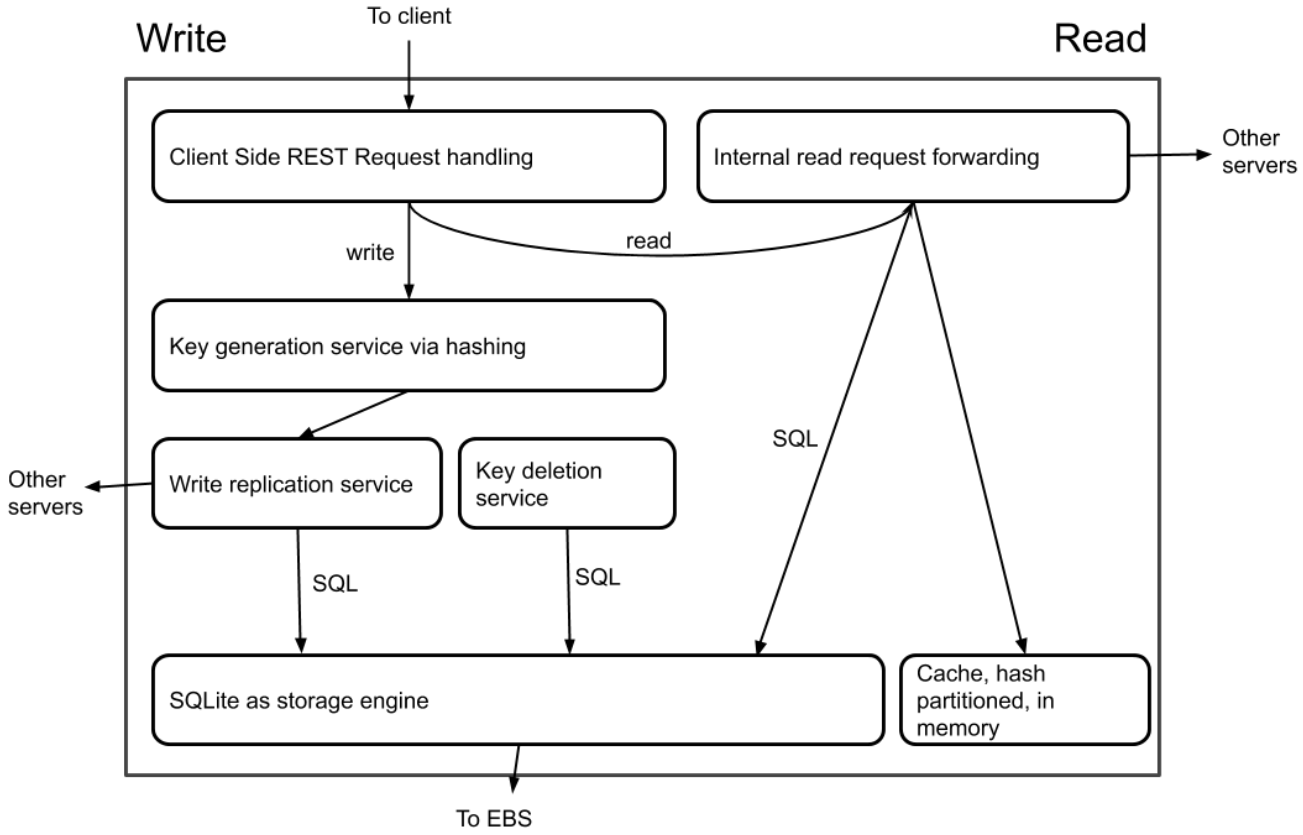

Figure 2: Detailed design of an EC2 instance

2.1 Read query execution

The process is described first without the caching layer and then with the caching layer in place.

Without the caching layer the process is rather simple:

1. node x receives a request from the client to return the URL for an alias
2. x looks up this key in its database
3. x returns the result to the client

With the caching layer, we need to forward the request to the correct node:

1. node x receives a request from the client to return the URL for an alias
2. if the alias is not cached on this node:
   (a) x hashes the alias and forwards it to the node holding the respective partition, in this case node y
   (b) y checks its in-memory cache and returns the result to x
   (c) if y does not find the key in cache, it looks it up in its database and returns the result to x
   (d) x returns the result to the client
3. else, if this is locally cached x can directly request its own cache and database.

The system should be made 3-2 read available by introducing a time out on the request to y. If the timeout triggers, we fall back to the first process without the caching layer.

2.2 Write query execution

We enumerate all steps of a write query below:

1. the client issues a request to to node x providing the long URL
2. x hashes the long URL and generates an alias by taking the first bits of the hash
3. x writes the new key to its database
4. if the write commits
   (a) it forwards the write to nodes y and z
   (b) upon confirmation from y and z it confirms the write to the client
5. else
   (a) we had a hash collision
   (b) we concatenate an integer, starting at 0, to the URL before hashing
   (c) continue at step 2

We argue that the write request at y and z commits all the time because the chance to generating the same keys concurrently on multiple nodes is close to 0.

The system as described becomes write unavailable when a single node fails. Optionally, you can implement a partly synchronous replication where only one node needs to confirm the write request. This gives you a 3-1 fault-tolerant system. The recovery would be manually by replaying the missed updates aided by the index on the creation date.

3 Implementation

The implementation has to be done in Java 1.8 or newer. It needs to build and compile using Maven. It has to run on Amazon EC2 instances with attached *Elastic Block Service* instances.

We recommend following the schedule below:

1. implement the system without caching on a single node and get it to run with the client on your local machine (first week)
2. write unit tests covering most of your system, also run the provided medium sized workload and confirm that your latency for a single operation is under 100 ms on average (first week)
3. run the same setup on two Amazon EC2 instances (client and server) (second week)
4. implement the system running on 3 nodes without the caching layer (second week)
5. implement the caching layer component (third week)

In the tutorial each week, address issues according to this schedule. If you want us to discuss a specific problem in the tutorial, please let us know by **Monday** 6 am.

Our model solution for the first week uses the `org.xerial.sqlite-jdbc` driver to connect to the database and `com.sun.net.httpserver.HttpServer` for connections to the client. We might need to replace the `HttpServer` with our own better performing solution later. In particular, a non-blocking server implementation as described here would be a great option. We recommend implementing your own `Client-Side REST Request handling` service using Java NIO as an optional addition to this exercise. The blog post, from the link above, comes with its own GitHub repo containing example source code.

Depending on how things go, we might add a fourth phase in which we kill a node and test the fault-tolerance of the system or more thoroughly examine the performance of the system.

3.1 The client

The client is provided as a GitLab repository and should be run on its own Amazon EC2 instance. It connects to a variable number of other EC machines and generates REST requests to read the URL given an alias and to write a new alias/URL combination. The interface is shown in listing 1.

```
GET /<alias> {}          ⇒ 200: {<url>\n}
                         ⇒ 404: {} for expired aliases
POST /       {<url>\n}   ⇒ 201: {<alias>\n}
```

Listing 1: Client requests and expected responses, the body of requests and responses is shown in {}

You find the client and usage instructions at GitLab.

3.2 Workloads

We provide two workloads: one to debug your system which uses a small dataset and runs only a few operations and one to test the system under load with a much bigger dataset and more operations to run. The datasets are described in subsection 3.3.

Both workloads are supposed to be run in the following steps:

1. Start all nodes which run your program, initialize them and connect them.
2. Bulkload the dataset file associated with the workload on all nodes.
3. Start the client specifying the workload and servers to run against.
4. Collect the log file from the client to evaluate.

The debug workload and one running more operations are available in the client repository. We provide the stresstest workload later.

3.3 Test datasets

We provide CSV files which contain different amount of aliases to load as a base dataset in your system. There are two sizes of datasets. First, a small one containing only 10 records for debugging purposes. Second, a medium sized one containing 9.25M records. The debug files contains two URLs which are expired or will during the exercise. Please, add tests for the deletion service yourself. We give an example of their format in listing 2.

```
0,2020-11-26 19:03:57.026,"!A5,Tk""8"
1,2020-11-26 19:03:57.048,+4448*&P
```

Listing 2: Example CSV data file

3.4 Logging

A distributed system is very hard to debug, e.g. it is not easy to use debuggers at multiple machines to step through the program. We strongly recommend writing unit test covering most of the system without needing to run more than one process. You can use mocking to simulate different events and responses from other systems. For bugs that do not arise in a local environment, the best approach is to use structured logs extensively. These allow you to retrace all important events after a bug occurred.

We recommend using a simple Log4j setup to trace all important events. For submission, you are required to provide a log file of the most important events anyhow. The log file needs to cover the following events:

1. Client related:
   (a) RECEIVED_CLIENT_REQUEST(<ID>,GET|POST,<params>) <threadid>
   (b) SEND_CLIENT_REPONSE(<ID>,GET|POST,<params>) <threadid>
2. Key generation service: GENERATED_HASH_FOR_URL(<hash>,<url>) <threadid>
3. Write replication service:
   (a) LOCAL_WRITE(<alias>) <threadid>
   (b) REMOTE_WRITE_REQUESTED(<other_nodeID>,<alias>) <threadid>
   (c) REMOTE_WRITE_RECEIVED(<other_nodeID>,<alias>) <threadid>
   (d) REMOTE_WRITE_CONFIRMED(<other_nodeID>,<alias>) <threadid>
4. Read request forwarding:
   (a) READ_FORWARDED(<other_nodeID>,<alias>) <threadid>
   (b) READ_CACHE_SUCESS(<alias>) <threadid>
   (c) READ_STORAGE_SUCESS(<alias>) <threadid>

Each event has to be reported together with a timestamp.

3.5 Alias generation

The client relies on the exact implementation of the alias generation service, so it knows which keys have been written and can be requested. Therefore, the alias generation should be done by taking the first 36 bits of the MD5 hash of the URL and encode them in Base64. Java source code is given in listing 3

```
String generate_alias(String url) {
    MessageDigest md5 = MessageDigest.getInstance("MD5");
    byte[] hash = md5.digest(url.getBytes());

    String key = Base64.getEncoder().encodeToString(hash);
    key = key.substring(0, KEY_LENGTH);
    if (key.startswith("/")) {
      key = key.replaceFirst("/", "=");
    }
    return key;
}
```

Listing 3: Alias generation code

3.6 Using the Java HTTP server with multiple threads

The `com.sun.net.httpserver.HttpServer` normally runs with a single thread. You should run it with multiple threads. Use the following source code:

```
        ExecutorService executorService = Executors.newFixedThreadPool(20);
        httpServer.setExecutor(executorService);
```

Listing 4: Running the HTTP server multithreaded.

3.7 Working with Amazon AWS

Amazon AWS grants students 100 Euro worth of credits. These should last you for this and the next exercise. Therefore, we recommend to run most of the testing on your machine and use the cloud only to test the performance of your final system. Do not forget to shut down instances when you complete your testing. If you use your resources carefully, 100 Euro should easily carry you through this exercise and the next one.

Please, use this link to signup. You might need to apply for an AWS Educate Starter account separately after signing in. The accounts are normally approved within a day. Report any issues you might encounter or if your account has not been approved within three days via Mattermost.

4 Submission Guidelines

Please hand in a link to a GitLab repository which contains your source code, a script to compile your source code, a script to run your system and the client locally in four different processes, a script to run your system on AWS and the log files of client and system after executing the stresstest workload. You can use the Gitlab of the chair All scripts should run on a Ubuntu 20.04 Linux machine. We describe the format of the server log-files is described on the next page. The expected folder structure your repo is shown in listing 5.

```
/compile.sh    // Compiles your code to a single JAR file
/run-local.sh  // Runs your system locally in four processes
               // with the debug workload
/run-aws.sh    // Runs your system in the AWS cloud,
               // instances to run out are provided as input parameters
/logs/
    client.log
    server-1.log
    server-2.log
    server-3.log
/src/<your source code>
```

Listing 5: ZIP file format

The events to report on each server have been described before in subsection 3.4. Please, do not include any other events and report in the format shown in listing 6.

```
[<hour>:<minute>:<sec>,<millisec>]    <event_type_with_params>\n
```

Listing 6: Log line format

We strongly recommend to finish most of the exercise before the Christmas break. Both of us are on holiday until the 10th of January 2021. Emails and Mattermost will be answered only occasionally during that period.