Milestone 3: Distributed & Replicated Storage Service

Cloud Databases (Due March 19th)

Overview

Availability and performance are vital prerequisites for large-scale Web data platforms. A prominent concept to achieve these properties is replication.

Replication introduces redundancy by storing items in disparate locations (usually physically distributed.) This increases availability, since a number of failing nodes can be tolerated. Furthermore, replication forms the basis for enhanced performance. Client requests can be distributed among different replicas and, consequently, the load on individual servers is reduced.

The objective of this milestone is to extend the storage service from Milestone 2 by means of replication. Data items (i.e., key-value pairs) are still distributed over a bunch of storage servers via consistent hashing. However, the storage servers are no longer just responsible for their own subset of the data, but also serve as replicas for data items of other servers. In this sense, a storage server assumes different roles for different data items. A storage server may be a:

- **coordinator**, if it is directly responsible for the data item, following the concept of consistent hashing detailed in the previous milestone. This means it has a position on the ring, such that it is the closest successor storage server in the ring topology according to the data item position (cf. Milestone 2).
- or a **replica**, if the data item is coordinated by another storage server and the data is simply replicated onto this storage server. It is either *replica_1* if it is the *first* successor of the coordinator or *replica_2* if it is the *second* successor.

In this milestone, each data item should be replicated on exactly two storage servers that are succeeding the coordinator on the ring topology. Replication is invoked and managed by the coordinator. The focus of this milestone is to implement a replication strategy that guarantees **eventual consistency**. Your strategy should also provide a **reconciliation mechanism for topology changes** that come about due to management actions invoked by the External Configuration Service (ECS) (i.e., adding or removing storage servers from the service), or due to crashes of individual storage servers. To detect failing servers and trigger the reconciliation mechanism, a **failure detection strategy** should be designed. Once failing servers are detected, ECS invokes the launch of a new storage server and reallocates data.

Learning objectives

With this milestone, we pursue the following learning objectives:

- Deal with a distributed system of enhanced complexity,
- detect and handle failures in distributed systems, including on client and server sides,
- understand the necessity for replication in large-scale distributed systems, and
- get exposed to the concept of eventual consistency.

Detailed milestone description

Assigned development tasks and deliverables

We recommend that on your first read of this milestone specification, you only glance at the deliverables below, then, read and understand the rest of the specification, before coming back and reading this deliverable section in more detail.

1. Design and implement the replication mechanism.
2. Design and implement failure detection and recovery within your service (among the storage servers) and ECS (ZooKeeper)
3. Design a mechanism that allows clients to gracefully handle the failure of the storage server they are connecting to or connected to.

- Automate the running of the given test cases
- Do regression testing, i.e., make sure that relevant Milestone 1 and Milestone 2 tests still pass
- Test functionality such as create connection/disconnect, get/put value, update value (existing key), get non-existing key (check error messages)
- **Add at least 10 additional test cases of your choice that cover the additional functionality and features of this milestone** (e.g., ECS extensions, replication, failure detection (client vs. server sides) etc.)
- Compile a short test report about all test cases, especially your own cases (submit as an appendix of your design document).
- Performance evaluation, data set, and metrics for testing (repeat your evaluation from Milestone 2 for your Milestone 3 design and implementation; compare your performance results from both milestones)
- Evaluate the performance of your storage service implementation. Use the **Enron Email data set (http://www.cs.cmu.edu/%7Eenron/)** to populate your storage service and run experiments. Measure latency and throughput for read

and write operations in varying scenarios. Evaluated at least the following:

- Different number of clients connected to the service (e.g., 1, 5, 20, 50, 100)
- Different number of storage servers participating in the storage service (e.g., 1, 5, 10, 50, 100)
- Different KV server configurations (cache sizes, strategies)

Also, consider evaluating the process of scaling the system up and down, i.e., how long does it take in the different scenarios to add/remove 1 ... *n* KVServers (you chose *n*). Your report should contain the quantitative results (e.g., plots) and an explanation of the results.

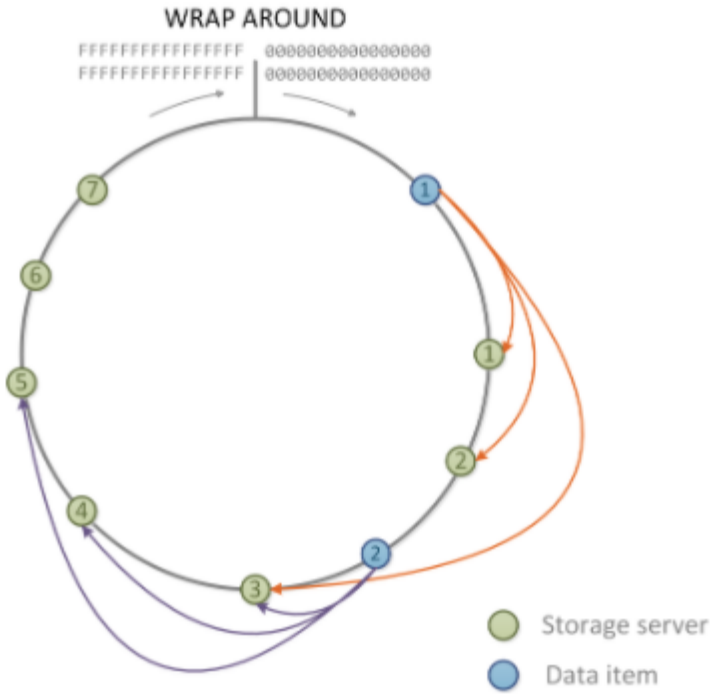
A **design document** that is no longer than 4 pages detailing your design (only report on the new elements added relative to the previous milestone), any design decisions you made and include the performance evaluation and report; the documentation of your test cases is not counted among the page count.

In addition to above requirements, you would be asked to start a client from a different machine during your demos. This might involve updating IP addresses and opening the required ports. A student could connect to servers running on another teammates' machine for testing.

Replication

Following the above described concept of coordinator and replica, data items are replicated to exactly two storage servers that succeed the data item's coordinating storage server on the ring topology. The below figure illustrates how data items are associated with storage servers (coordinator and replicas.)

You will need to extend your ECS (i.e., extend the use of ZooKeeper) and KVServer logic to implement replication that guarantees eventual consistency.



Eventual consistency

Eventual consistency is a weaker form of consistency and mostly suitable for read-dominant systems. It ensures that, provided there are no updates for a “long” time, all replicas will gradually become consistent (i.e., eventually converge to the same, consistent state.)

System reconciliation under scaling

To maintain the replication invariant (i.e., every data item is replicated on the two successor storage servers of the coordinator), even when adding or removing storage server from the storage service, the ECS must initiate the reassignment and reallocation of replicated data, not only on the coordinator, but also on replicas. Design a mechanism to implement this behavior. The ECS should be involved as little as possible.

Failure detection & reconciliation

Similar to the behavior under up/down scaling, the system should maintain the replication invariant, also in the case of failure (i.e., crashing storage servers). To let the ECS detect such crashes, implement a failure detection strategy. Again, design a mechanism that involves the ECS as little as possible (in terms of the communication and request rate it would have to be subjected to.) Could you leverage ZooKeeper for failure detection?

When a failed storage server is detected, ECS does two things:

1. First, the storage service is reconstructed according to the new set of available storage servers (i.e., recalculation of data serving responsibilities based on consistent hashing, update of metadata at storage servers, and reallocation of data items on coordinator and replica.)
2. Second, the crashed storage server is replaced by a new storage server (use and add addNode() to ECS for this purpose.)

Guidelines for implementation

Within this milestone, you have the opportunity to make most design decisions on your own.

However, you should adhere to the following guidelines:

- Write requests need to be served by the coordinator for a given data item, whereas read requests may also be served by any replica. Note, since the implementation should only guarantee eventual consistency, the result value retrieved may be stale).
- Replication is invoked and managed by the coordinator. Note, that the coordinator also has to manage deletion of data and updates to data, i.e., if a data item has been deleted, the coordinator also has to ensure deletion of the data on all replicas.

Suggested development plan

You should base design and implementation of your replication strategy on your design for Milestone 2.

Getting the project files

There are no new project files for Milestone 3. Base your project on your Milestone 2 implementation and the project files distributed for Milestone 2.

Deliverables & code submission

By the **deadline**, you must hand in your software artifacts that implement all the coding requirements and include all necessary libraries and the build script. See the above specified deliverables for what needs to be submitted. **Submission instructions are same as before (https://q.utoronto.ca/courses/202854/pages/submission-instructions-and-normalized-environment-same-as-for-milestone-1)**

Please submit the report on **Quercus M3 Assessment (https://q.utoronto.ca/courses/202854/assignments/574981?module_item_id=2391944)** by deadline.

Additional resources

- Java Coding Conventions: **http://www.oracle.com/technetwork/java/codeconv-138413.html (http://www.oracle.com/technetwork/java/codeconv-138413.html)**
- Log4j: **http://logging.apache.org/log4j/2.x/ (http://logging.apache.org/log4j/2.x/)**
- JUnit: **http://www.junit.org/ (http://www.junit.org/)**
- Ant build tool: **http://ant.apache.org/ (http://ant.apache.org/)**

Document revisions

Changes to the milestone handout after posting it are tracked here.

Date	Change
17 March 2021	Submit report on Quercus