DIDA-GSTORE

Design and Implementation of Distributed Applications 2020-2021
IST (MEIC-A / MEIC-T / METI)
Project Statement

1 Introduction

The goal of this project is to design, implement and evaluate a simplified version of a geo-replicated storage system. The system is composed of two types of nodes: servers and clients. The servers, collectively, keep a distributed key-value store. The clients perform read and write operations. Write operations submit a key and a value and store both in the storage system. Read operations submit a key to the system and return a value.

Servers are located in different geographical regions (in a real system, servers would actually be data centers).

2 Overview

Each object in the data store is identified by a unique key and stores a value. The unique key consisting of a tuple that includes two fields, namely, a partition identifier and an object identifier: $\langle partition\_id, object\_id \rangle$. For simplicity, we assume that the value is just a string with some maximum size (in a real system, it could be an arbitrarily large object). Objects can be *written* (when an object is written, the string is overwritten, i.e., the old string is deleted and the new string is kept) and *read* (the current string is returned to the client).

3 System Architecture

The system supports partial replication. Each partition is stored on a pre-defined set of servers. Different partitions are stored on different sets of servers. One of the servers is considered the master replica for that partition. For instance, consider a system with 5 servers ($S_1, S_2, \ldots, S_5$), and 5 partitions ($P_1, P_2, \ldots, P_5$) replicated in 3 servers each. A possible deployment could be:

- P_1 is stored on servers S_1, S_2, S_3 with S_1 as the master;

- P_2 is stored on servers S_2, S_3, S_4 with S_2 as the master;
- ...
- P_5 is stored on servers S_5, S_1, S_2 with S_5 as the master.

Assume that the mapping between partitions and nodes is known to all participants in the system, both servers and clients.

When a client performs the first *read* operation on a given object, it *attaches* itself to a server that replicates that object. When doing further *reads*, the client will remain attached to that server, as long as the server replicates the objects the client needs to read. If, at some point, the client needs to read an object that is not replicated on the attached server, the client will *detach* and *attach* to a new server that replicates the desired object. To perform READ operations, the client can be attached to any replica. To perform a WRITE operation, the client needs to be attached to the master server (for that partition). So, in order to do a write operation, the client may need to attach to another server, even if the current server replicates the object.

To simplify the project, servers should store the data *exclusively* in volatile memory.

Students will need to implement two versions of the system.

3.1 Base Version

The first version offers *linearizability*[2] and works as follows:

- WRITE operation: The master servers for each partition serve write requests one at the time. If a new request arrives while another write request is executing, the request is queued and served only when the current write operation terminates. To execute a write, the master sends a lock request to all replicas and waits for an acknowledgement from every replica. Then, the master send the new value to all replicas. When a replica receives the new value, it applies the value and unlocks the object and sends a reply back to the master. When the master receives the second round of replies, it returns to the client.
- READ operation: read operations can be executed at any replica. When a server receives a read request, if the object is unlocked, the server immediately returns the current value. If the object is locked, the read request is queued and served when the object becomes unlocked.

3.2 Advanced Version

The previous algorithm offers strong consistency but has many drawbacks, namely:

- Operations have a strong latency (6 communication steps);

- If the master fails, writes can no longer be performed. Also, if the master fails before unlocking the object, reads can also no longer be performed.

Students should design an alternative version that offers high availability and that:

- makes operations more efficient;
- reduces or eliminates the chances that a client becomes blocked.

To achieve this, the consistency model offered by the storage system may need to be weakened. That is, instead of offering linearizability, the system should support some other consistency criteria. It is up to the students to decide which consistency criteria they want to support. Students should justify their choice.

3.3 Fault Tolerance

The reason for having multiple replicas of each partition is to provide fault-tolerance and lower latency on read operations. If a replica fails, the system's functionality will be still available at other replicas.

The project will make some simplifying assumption regarding the occurrence of faults, namely:

- We assume that replicas can only fail by crashing and that, when a replica fails this fact can be reliably detected by other nodes (in other words, you can assume the availability of a perfect failure detector). Thus, when a replica fails, all the other nodes are eventually informed of the failure and can update the view of active servers accordingly.
- A node that crashes never recovers.
- Also, no new nodes join the execution.
- The network is not subject to partitions.

Under these assumptions, the system should provide strong guarantees on the reliability of the distributed execution and liveness in the presence of failures (assuming that the maximum number of faults is bounded).

Note that, in a real system, network partitions could occur and, in most cases, reliable failure detection will not be realistic. Thus, a real system would need to consider the limitation of the CAP theorem[1] into consideration and algorithms such as Paxos[3].

4 Implementation

The project should be programmed using C# and use G-RPC for remote communication.

4.0.1 Client Scripts

Clients execute a sequence of commands contained in an input script. The client scripts are text files submitted to the client as a command line parameter. They are assumed to be available on the client machines on the same directory as the client program. The scripts should be executed synchronously. The commands a script may contain are:

- `read` *partition_id object_id server_id*:
  Reads the object identified by the $\langle partition\_id, object\_id \rangle$ pair and outputs (on screen and eventually to a log) the corresponding value. It should return the string "N/A" if the object is not present in the storage system. If the server that the client is currently attached to does not hold the requested object, the client will use the *server_id* parameter to try to find the object at the *server_id* server. If the client does not need to change server to obtain the requested object or if the *server_id* parameter is -1, the parameter should be ignored.

- `write` *partition_id object_id value*:
  Writes the object identified by the $\langle partition\_id, object\_id \rangle$ pair and assigns it the quotes delimited string *value* (e.g. "a possible stored value").

- `listServer` *server_id*:
  Lists all objects stored on the server identified by *server_id* and whether the server is the master replica for that object or not;

- `listGlobal` :
  Lists the partition and object identifiers of all objects stored on the system.

- `wait` x :
  Delays the execution of the next command for x milliseconds.

- `begin-repeat` x :
  Repeats x number of times the commands following this command and before the next `end-repeat`. It is not possible to have another `begin-repeat` command before this loop is closed by a `end-repeat` command. Within the repeat cycle any occurrence of the string "$i" should be replaced by the number of the iteration of that cycle being performed, from 0 to $x - 1$. For example, the folowing three lines cause the insertion of three objects called "obj-1", "obj-2" and "obj-3":
  ```
  begin-repeat 3
  write part-1 obj-$i "value-$i"
  end-repeat
  ```

- `end-repeat` :
  Closes a repeat loop.

4.1 Simplifying the Fault-tolerant Code

For this project, it is assumed that at most f faults may occur, where $f < MinPartitionSize$, being *MinPartitionSize* the number of servers of the partition will the smallest number of replicas, such that a partition is never lost. Furthermore, the students can assume that only one fault may happen at each moment and that the system will have time to recover before the next fault. Obviously, how much time is required for that recovery will be taken into account in the project grading.

4.1.1 Message Delivery Delays

The implementations should ensure that it is possible to arbitrarily delay the arrival of a message at a server. When servers are started they will should be configured to delay any incoming message for a random number of milliseconds chosen from a given interval (see Section 5 below). It can be assumed that messages between two nodes are never reordered, i.e. the channels between nodes guarantee FIFO (first in, first out) ordering.

5 PuppetMaster

To simplify project testing, all nodes will also connect to a centralised process called the *PuppetMaster*. The role of the PuppetMaster process is to provide a single console from where it is possible to control experiments. Each physical machine used in the system (except for the one where the PuppetMaster is running) will also execute a process, called PCS (Process Creation Service), which the PuppetMaster can contact to launch processes (clients or servers) on remote machines. Once a process is created (server or client), it should interact with the PuppetMaster directly. For simplicity, the launching of the PuppetMaster and of the PCS will be performed manually. Since the servers and clients are processes (receiving configuration command line parameters), it should be, in principle, possible to alternatively operate the system without the need for a PuppetMaster or PCS.

It is the PuppetMaster that reads the system's configuration file and starts all the relevant processes. The PCS on each machine should expose a service at a URL on port 10000 for the PuppetMaster to request the creation of a process. For simplicity, we assume that the PuppetMaster knows the URLs of the entire set of available PCSs. This information can be provided, for instance, via configuration file or command line. The PuppetMaster can send the following commands to the nodes in the system:

- `ReplicationFactor` *r*: configures the system to replicate partitions on *r* servers.
- `Server` *server_id URL min_delay max_delay*: This command creates a server process identified by *server_id*, available at *URL* that delays any incoming message for a random amount of time (specified in milliseconds) between *min_delay* and

max_delay. If the value of both *min_delay* and *max_delay* is set to 0, the server should not add any delay to incoming messages. Note that the delay should affect *all* outgoing communications from the server.

- `Partition` *r partition_name server_id_1 ... serverd_id_r*: configures the system to store *r* replicas of partition *partition_name* on the servers identified with the server_ids *server_id_1* to *serverd_id_r*.

- `Client` *username client_URL script_file*: This command creates a client process identified by the string *username*, available at *client_URL* and that will execute the commands in the script file *script_file*. It can be assumed that the script file is located in the same disk folder as the client executable.

- `Status`: This command makes all nodes in the system print their current status. The status command should present brief information about the state of the system (who is present, which nodes are presumed failed, etc...). Status information can be printed on each nodes' console and does not need to be centralised at the PuppetMaster.

Additionally, the PuppetMaster may also send debugging commands to the server replicas:

- `Crash` *server_id*. This command is used to force a process to crash.

- `Freeze` *server_id*. This command is used to simulate a delay in the process. After receiving a freeze, the process continues receiving messages but stops processing them until the PuppetMaster "unfreezes" it.

- `Unfreeze` *server_id*. This command is used to put a process back to normal operation. Pending messages that were received while the process was frozen, should be processed when this command is received.

The PuppetMaster script starts with the server setup (Server and Partition commands). All partitions may have the same replication factor. The PuppetMaster should have a simple console, preferably with a GUI, where a human operator may type the commands above, when running experiments with the system. Also, to further automate testing, the PuppetMaster should also be able to read a sequence of such commands from a *script* file (whose file name is input in the PuppetMaster GUI) and execute them sequentially or step by step. A script file can have one more additional command that adjusts the behaviour of the PuppetMaster itself:

- `Wait` *x_ms*. This command instructs the PuppetMaster to sleep for *x* milliseconds before reading and executing the next command in the script file.

For instance, the following sequence in a script file will force a server *broker0* to freeze to 100ms:

```
Freeze server_url
Wait 100
Unfreeze server_url
```

All PuppetMaster commands should be executed asynchronously except for the `Wait` command. For example, the PuppetMaster should return from the server creation command as soons as possible and not wait for the system membership to stabilize. If necessary, those wait steps will be added using `Wait` commands when testing. Port 10001 should be reserved for the PuppetMaster and can be used to expose a service that collects information from the system's nodes when the Status command is done.

6 Performance Evaluation

Each group must evaluate the system's performance by identifying the workloads for which the implementation performs best and worst and design clients that test those situations as well as baseline scenario. The resulting performance data should be discussed in the report (see next section).

7 Final Report

Students should prepare a final report describing the developed solution (max. 6 pages). In this report, students should follow the typical approach of a technical paper, first describing very briefly the problem they are going to solve, the proposed implementation solutions, and the relative advantages of each solution. Please avoid including in the report any information already mentioned in this project description. The report should include an explanation of the algorithms used and justifications for the design decisions. The final reports should be written using LaTeX. A template of the paper format will be provided to the students.

8 Checkpoint and Final Submission

The grading process includes an intermediate optional checkpoint and a mandatory final submission. For the checkpoint the students may submit a preliminary implementation of the project; if they do so, they may gain a bonus in the final grade. The goal of the checkpoint is to control the evolution of the implementation effort. Given that students are expected to perform an experimental evaluation of the prototype, it is desirable that they have a working version of the base version that supports simple crashes by the checkpoint submission deadline. In contrast to the final evaluation, in the checkpoint only the functionality of the project will be evaluated and not the quality of the solution.

After the checkpoint, the students will have time to add the advanced version of the system plus any missing fault tolerance mechanisms, perform the experimental evaluation and fix any bugs detected during the checkpoint. The final submission should include the source code (in electronic format) and the associated report (max. 6 pages). The project *must* run in the Lab's PCs for the final demonstration.

9 Relevant Dates

- November 5th - Electronic submission of the checkpoint code;
- November 9th to November 13th - Checkpoint evaluation;
- December 4th - Electronic submission of the final code;
- December 7th - Electronic submission of the final report.

10 Grading

The project grading will depend on a discussion at the end of the semester where all members of the groups must be present and where individual grades will be determined. That grade will depend on, besides the quality of the project, the individual performance in the discussion and the lecturer's evaluation.

The project grade (45% of the course's grade) is the *best* of the following two:

- Final_Project_Grade
- 85% of the Final_Project_Grade + 15% of Checkpoint_Grade

11 Cooperation among Groups

Students must not, *in any case*, see the code of other groups or provide their code to other groups. If copies of code are detected, all groups involved will fail the course.

12 "Época especial"

Students being evaluated on "Época especial" will be required to do a different project and an exam. The "Época especial" project will be announced on the first day of the

"Época especial" period, must be delivered on the day before last of that period, and will be discussed on the last day.

References

[1] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[2] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[3] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.