DIDA-2021

Design and Implementation of Distributed Applications 2021-2022
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 distributed function-as-a-service cloud platform. This platform will run applications composed of a chain of custom operators that share data via a storage system. Applications will be run by sending a script (that specifies the chain of operators to be executed) to a scheduler process. The scheduler starts by assigning a computing node (a *worker*) to each operator, and then instructs the worker running the first operator to initiate the execution of the chain. The application operators then call each other in a daisy chain sequence. The application data is stored in a storage system maintained by a set of storage nodes.

The design and implementation of the project will have to address several challenges raised when building such a distributed system such as: how to access and replicate the application data records, where to execute each operator, and how to maintain data consistency between the system's storage nodes.

2 System Architecture

The **DIDA-2021** platform is composed of a set of computing nodes where applications are run, a set of storage nodes where data is stored, a scheduler used to deploy and start applications, and a control node, called the *PuppetMaster*, used to test and debug the system. To simplify the implementation, clients are not executed explicitly; the PuppetMaster instructs the scheduler to execute scripts *as if* they had been sent by real clients.

Applications in **DIDA-2021** are composed of a sequence of operators. Each function operator receives a input metarecord (contained in a request), executes a user-provided function and outputs a result that is included in the request forwarded to the next operator in the application chain.

Operators can be started on any computing node. When they are called by the previous operator on the application's operator chain, they receive a metarecord, an input

string and the output of the previous operator and do some processing based on those inputs.

Inputs to the application are stored in the request and the outputs of each operator are also stored in the request (and passed to the downstream operator in this way). Furthermore, operators can read and write from stable storage; this provides an indirect way for different operators to communicate (in addition to the output record stored in the request).

2.1 Storage

The storage nodes in the **DIDA-2021** platform store the data processed by the computing nodes. The system supports partial replication. Each data record is stored on a predefined set of servers (a subset of the full set of storage servers). All data items should initially have the same replication factor. Output records are synchronously written to one of the replicas which, in turn, will disseminate the update to other replicas in background, using a gossip protocol.

Storage is multiversioned, i.e., multiple version of the same item are kept. A version identifier is a tuple in the form:

```
class DIDAVersion {
    int versionNumber;
    int replicaId;
  }
```

When an item is written a new version identifier is created, by assigning it a version number larger than any other version number of that item that is known by the replica that executes the update. The id of the replica that executes the update is also recorded in the version identifier and used to order concurrent updates, that may be applied by different replicas (and be assigned the same version number). *DIDAVersion*s can therefore be totally ordered. We assume that each replica keeps at most *MaxVersions* versions of each item (discarding the oldest version if a new version is created and this number is exceeded).

The interface of the storage system is as follows:

```
  interface IDIDAStorage {
    DIDARecord read(string id, DIDVersion version);
    DIDAVersion write(string id, string value);
    DIDAVersion updateIfValueIs(string id, string oldvalue, string newvalue);
  }
```

```
public class DIDARecord {
  string id;
  DIDAVersion v;
  string val;
}
```

When writing an item, the version of the update is returned to the client. When reading, the client can specify that it wants to read a specific version. If a *null* version is provided as input when reading, the storage simply returns the most recent version. If the client specifies a given version that is not available at the target replica, a *null* value is returned.

The storage also provides a primitive *updateIfValueIs* that is conceptually close to *compareAndSwap* This primitive works like a write, but the write can only be applied if the value of the register matches a certain specified *oldvalue*. For instance, *updateIfValueIs("box", "empty", "full")*, will only update the value of the *"box"* if its current value is *"empty"*. An execution of *updateIfValueIs* must be totally ordered with respect to all other executions of *updateIfValueIs* and *write* operations and, when the value of the register is tested against *oldvalue*, all previous conditional updates and writes must have been applied. When the *updateIfValueIs* succeeds, a new version is returned to the client, otherwise a null version is returned.

It is assumed that the mapping between data identifiers and servers (i..e, which servers replica each item) is known to all nodes in the system, including the scheduler and all workers. Students may use consistent hashing to define the mapping or use some other solution they think will provide better results.

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

2.2 Applications

Applications are specified as a sequence of commands contained in an text file. Each command in an application file has the following format:

- `operator` *classname seq_no*:
  Adds an instance of the class *classname* as an operator in the application. This operator will be placed in the position *seq_no* in the application chain of operators. It is assumed that the operators in an application follow a continuous sequence of operators starting at 0.

2.3 Client Requests

A client request is a tuple (string input, string application_file). Client requests are received by the scheduler that sets up the required operators on worker nodes and sends

a `DIDARequest` (see below) to the first operator.

2.4 Scheduler

Each execution of a given application is assigned an unique id when it is started by the scheduler. This information is passed from worker to worker and it is called the *DIDAMetaRecord*:

```
class DIDAMetaRecord {
    int id;
    // other metadata to be specified by the students
  }
```

The students may add other fields to the *DIDAMetaRecord*, that can be updated as the application evolves (i.e., by each operator of the chain).

Also, the scheduler assigns a worker to each operator in the application script by creating an assignment object with the following format:

```
class DIDAAssignment {
    DIDAOperatorID    operator;
    string           host;
    int              port;
    string           output;
}

struct DIDAOperatorID {
string classname;
int order;
}
```

The assignment object registers the host name and port of the worker that has been assigned to execute the given operator. This object has also an *output* field that can be used by the operator to store some output. Finally, the scheduler creates a request object that will be passed from worker to worker during the execution of an application:

```
class DIDARequest {
    DIDAMetaRecord      meta;
    string               input;
    int                  next;
    int                  chainSize;
    DIDAAssignment[]      chain;
  }
```

where *next* indicates the next operator to be executed (initially, it is set to 0, the first entry of the chain).

2.5 Workers

Workers receive requests with the format described above and execute the following pseudo-code:

```
request.chain[next].operator.ProcessRecord(DIDAMetaRecord, inputString,
    previousOperatorOutput);
request.next = request.next +1;
if (request.next < request.chainSize)
    forward request to (request.chain[next].host, request.chain[next].port);
```

An operator receives a *DIDARequest* as input and can: i) read and update the *meta* field; ii) read the original *input* field and the *output* field of the previous operator (if any); iii) update its own *output* field. Operators can read and write to the data storage.

2.6 Data Consistency

The system must ensure that an operator, when accessing the data store, reads a consistent version of stored items. In particular, if an upstream operator has written an item that is subsequently read by a downstream operator of a given application, the downstream operator must read the version written by the upstream operator. Also, if two operators of the same application read the same item, they must both read the same version of that item. This can be achieved by repeating requests to the same replica or by contacting different replicas until the desired version is read. If an application tries to read a version that has been garbage-collected it should terminate.

2.7 Fault Tolerance

The reason for having multiple storage replicas is to provide fault-tolerance and distribute the load of read operations. If a replica fails, the data will likely be still available at other replicas. Note that, with the suggested implementation (where writes are applied to a single replica and propagated in background), updates can still be lost if that replica fails before propagating the update.

It is assumed that the scheduler and the computing nodes do not fail.

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. 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.

2.7.1 Simplifying the Fault-tolerant Code

For this project, it is assumed that at most f faults may occur, where $f < MaxVersions$. This ensures that there is at least one replica where writes and reads can be executed.

3 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 (workers and storage 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 a script with the system configurations 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 configuration script may contain the following commands:

- `scheduler` *server_id URL*
  Creates the scheduler process identified by *server_id*, available at *URL*.
- `storage` *server_id URL gossip_delay*
  Creates a storage process identified by *server_id*, available at *URL* that delays any gossip propagation messages for *gossip_delay* milliseconds.
- `worker` *server_id URL gossip_delay*
  Creates a worker process identified by *server_id*, available at *URL* that delays any request propagation message for *gossip_delay* milliseconds.

- `client` *input app_file*
  Issues a request to run the application described in *app_file* with the input string *input*.

- `populate` *data_file*
  This command inserts each line of the *data_file* as an item into the storage system. Each line is a comma-separated pair with a string identifier and a string value. Neither the identifier nor the string may contain commas or spaces.

- `status`
  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.

- `listServer` *server_id*
  Lists all objects stored on the server identified by *server_id*.

- `listGlobal`
  Lists all objects stored on the system.

It may be assumed that a configuration file begins with a `debug` command (if it exists) followed by a sequence of `scheduler`, `storage` and `worker` commands to set up the system. Any additional commands will appear after this setup. In addition to the commands above, the PuppetMaster script may also contain debugging commands to be sent to the system processes:

- `debug`
  Starts the system in debug mode and makes all worker nodes log their outputting of data or metadata records to the PuppetMaster. This command can be assumed to appear always at the beggining of the configuration script.

- `crash` *server_id*. This command is used to force the storage process *server_id* to terminate in order to simulate a fail-stop crash.

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

The PuppetMaster script starts with the servers setup (storage and worker commands). 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.

All PuppetMaster commands should be executed sequentially but asynchronously except for the `wait` command. For example, the PuppetMaster should return from the server creation command as soon 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 logging the output in debug mode.

4 Implementation

The project should be programmed using C# and use gRPC for remote communication. For simplicity, it can be assumed that the operator code is available on all nodes. Operator classes should not be linked be loaded dynamically via reflection. It is also important to note that, although the scripts syntax should not be altered, additional parameters can sent in the communication between servers. For example, when the PuppetMaster starts a worker node it could send it the list of all storage servers.

5 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).

6 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.

7 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, the checkpoint grade may improve their final grade. The goal of the checkpoint is to control the evolution of the implementation effort.

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.

8 Relevant Dates

- October 22^{th} - Electronic submission of the checkpoint code;
- October 25^{th} to October 29^{th} - Checkpoint evaluation;
- November 12^{th} - Electronic submission of the final code;
- November 12^{th} - Electronic submission of the final report.

9 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

10 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. An exception to this rule is made for application operators (not the worker node code!): groups may share with other groups the applications they create to test the platform.

11 "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.