PADI-FS

PADI
Plataformas para Aplicações Distribuídas na Internet
Middleware for Distributed Internet Applications
Project - 2012-13

MEIC-A / MEIC-T / MERC / EMDC
IST

Abstract

The PADI project aims at implementing a simplified distributed file system.

1 Introduction

The goal of this project is to design and implement **PADI-FS**, a distributed system to manage sets of files. Users can develop applications that access the **PADI-FS** to keep their files replicated and highly available. The relevant operations on files are limited to open, create, read, write, close and delete.

This distributed store is inspired by, and a simplification of, real systems such as Google File System, or HDFS. The contents of the files are backed by a number of data servers that provide their storage space, and are coordinated by the clients.

The work will start by writing a paper describing the solution (see Sec. 8) and only then should the system be developed sequentially in phases (see Sec. 9).

2 Architecture

The **PADI-FS** architecture is composed of the following components, that will correspond to four types of processes:

- *Puppet Master*: A centralized controller that can issue commands to all other components. The purpose of this component is to simplify testing and debugging.
- *Metadata Server*: A server that keeps metadata about the files. The Metadata Server is replicated, and this is implemented by 3 replicas. The replicas are assumed to use a dedicated network, such that the failure of a replica can be reliably detected by the remaining replicas.
- *Data Server*: A server that stores the contents of files. There are many Data Servers and the content of a given file may be replicated in several Data Servers. Data Servers may be placed in different places on the internet, and may be subject to temporary disconnections.
- *Client Process*: A process that makes requests to create, open, close, delete, read and write files.

All processes except the Puppet Master export at least two interfaces:

- An interface providing services needed for **PADI-FS**.

- An interface to receive commands from the Puppet Master.
- The Puppet Master can receive commands from the console or read commands from a file and send them to the other components.

2.1 Data Server

Each Data Server stores the content of one or more **PADI-FS** files. The contents of each **PADI-FS** file is stored using the local file system, in a local file with a name that is generated by the Metadata Server. Local files are all stored in the same directory and are named using 16-character ASCII strings. The state of the Data Server should be written to disk to allow reloading the previous state of the Data Server when it is started.

Each file includes two fields:

- A version number (a monotonically increasing number, that starts with 0).
- A byte array (the contents of the file).

A Data Server receives requests of the following types directly from clients:

- READ (LOCAL-FILE-NAME, SEMANTICS): returns the version and the content of the local file. The purpose of the semantics will be explained later.
- WRITE (LOCAL-FILE-NAME, BYTE-ARRAY): overwrites the content of the local file, creating a new version.

Furthermore, a Data Server also receives the following commands from the Puppet Master:

- FREEZE: it starts buffering read and write requests, without answering (simulates a network delay).
- UNFREEZE: responds to all buffered requests from clients and re-starts replying to new requests.
- FAIL: the data server starts ignoring requests from the clients or messages from metadata servers (simulates a failure).
- RECOVER: the data server starts receiving requests from the clients and from metadata servers (simulates a recover).

2.2 MetaData Server

The metadata server maintains a table, where each line stores relevant information about each file in the system. The information stored for each file is as follows:

- FILE NAME: a string.
- NBDATA SERVERS: the number of data servers used to store the content of the file.

- ReadQuorum: The minimum number of data servers that should be contacted during a read operation.

- WriteQuorum: The minimum number of data servers that should be contacted during a write operation.

- Data Servers: a list of pairs (data-servers, local-files) where the content of the file is stored.

This information should be also stored on disk, such that it can be reused when a metadata server recovers. This also simplifies testing and debugging, as it makes possible to start an experiment from a previously "populated" file system.

The metadata server receives the following requests from clients:

- Open (filename): returns to the client the contents of the metadata stored for that file (the entire data structure described above).

- Close (filename): informs the metadata server that the client is no longer using the file. The client must discard all the information about the data and acquire the information again via the Open if it needs to access the file later.

- Create (filename, nb-data-servers, read-quorum, write-quorum): it creates a new file (if it does not exist), by selecting the data servers that will be used to store the data, and by assigning an unique local file name on each of these data servers. In case of success, it returns the same information as in open.

- Delete (filename): deletes the file with the given name.

The metadata server, and each of its individual replicas, also receive the following commands form the Puppet Master:

- Fail: the metadata server stops processing requests from the clients or messages from other metadata servers (replicas).

- Recover: the metadata server starts receiving requests from the clients and from other replicas.

2.3 Clients

The client creates, opens, reads, writes, closes and deletes files as instructed by the Puppet Master.

When a client is created, it is provided with the contact of all metadata server replicas. A client can contact one of the metadata servers to Create, Open, Close or Delete a file. In particular, as a result of an open, it obtains information about the data servers where the content of the file is stored (and the names of the local files, at each of these servers, where the content is locally stored).

In order to perform Reads and Writes the client contacts the data servers directly:

- When performing a WRITE, the client must block until it receives a confirmation from a write quorum of data servers. The size of this quorum was defined in the file metadata when the file was created initially.
- When performing a READ, the client must block until it receives a confirmation from a read quorum of data servers. The size of this quorum was also defined in the file metadata when the file was created initially. The client returns the content associated with the response with the most recent version number.

3 Fault Tolerance

In the basic implementation, metadata servers never fail. Data servers may fail but eventually recover. READ and WRITE operations should terminate as long as the read quorum (resp. write quorum) is alive. If a quorum cannot be reached, the operation blocks until the required number of data servers reply to the request.

4 Concurrency Control and Consistency of the Metadata

As each metadata server replica may be responding to several client requests in parallel, they will be inherently concurrent and multithreaded; in each metadata server replica, multithreading must be taken into account. Students should synchronize the internal operation of each metadata server replica appropriately.

Furthermore, students are encouraged to design and research the possible alternative approaches and algorithms to ensure the overall consistency of metadata server tables, in the presence of replication and concurrent requests. Relevant issues include deciding on whether to allow any replica to handle client requests or forward them all to a primary replica. This entails different choices regarding consistency that students must ponder and address in their paper.

5 Session Guarantees

As noted before, the READ operation has one parameter named SEMANTICS. This parameter can take only two values: *default* and *monotonic*. A default read returns the most recent version of a read quorum. However, with this policy, it is possible that a client observes a given version (say version n) as the result of a read operation and subsequently, makes another read from the same file and obtains an old version of the data (say, version $n-1$). When the *monotonic* semantics is used, this should be prevented: clients must always read a version equal or higher than the version they have read in the past.

6 Interactions among Participants

This section describes how the different participants interact with each other.

Puppet Master In a real system, each client would have its own user interface, possibly controlled by a different (human) user. Unfortunately, this would make testing and debugging very cumbersome, as one would need to keep multiple windows open to control the different clients.

Therefore, in order to simplify debugging and evaluation, there should be a *puppet master* process, which partially controls the behavior of the remaining processes. The puppet master should offer a simple user interface that allows controlling the actions of every process in the system. The puppet master is the first process to be started in each execution of the system.

As a result, the client processes and the servers do not need to have a user interface. They should, instead, offer a remote invocation interface that the puppet master will use. In this way, complex executions can be controlled from a single interface, therefore simplifying debugging and testing. Servers should offer two additional remote invocation interfaces: one for other servers and one for clients.

Besides the interactive interface, the puppet master must have GUI commands to support the opening, running, and step-by-step running of execution trace files (e.g. buttons for "load script", "run loaded script" and "next step in loaded script"). An execution trace file is an ASCII text file where each line is a command as described in the previous section. The puppet master must support the commands listed below. All these commands have a parameter named PROCESS, which is used to identify a metadata server, a data server, or a client. Metadata servers are named `m-`n (with n equal to 0,1 or 2), data servers are named `d-`n (where n is the number of the Data Server), and clients are named `c-`n.

- FAIL PROCESS, which temporarily disconnects a process from the network, disabling its ability to send to or receive messages from other processes;
- RECOVER PROCESS, opposite of the above command;
- FREEZE PROCESS, similar to fail but the messages are buffered;
- UNFREEZE PROCESS, similar to recover, but the target processes buffered nessages before resuming normal operation;
- CREATE PROCESS, FILENAME, NbData Servers, ReadQuorum, WriteQuorum: to instruct the creation of a file with the corresponding metadata, and stores that information in a free file (structured) register in the Puppet Master.
- OPEN PROCESS, FILENAME: to open the desired file, where the corresponding metadata obtained is also stored in a free file (structured) register in the Puppet Master.
- CLOSE PROCESS, FILENAME: instructs a client to close a file that it has previously opened.
- READ PROCESS, FILE-REGISTER, SEMANTICS, STRING-REGISTER: reads the contents of the file identified by a file register and stores it in a string register in the Puppet Master.
- WRITE PROCESS, FILE-REGISTER, BYTE-ARRAY-REGISTER: writes the file identified by a file register with the contents previously stored in a string register of the Puppet Master.

- WRITE PROCESS, FILE-REGISTER, CONTENTS: writes the file identified by a file register with the contents of a string (to be stored as the byte array) provided in the command.
- COPY PROCESS, FILE-REGISTER1, SEMANTICS, FILE-REGISTER2, SALT: reads the content of file whose metadata is stored in file-register1, and writes as new contents of the file with metadata in file-register2, the concatenation of the content of the first file with a string (to be stored as a byte array), that serves as salt to make them slightly different.
- DUMP PROCESS: prints all the values stored at the metadata servers, including for each file, its metadata, and the data servers holding replicas of the file, with the corresponding local filenames.
- EXESCRIPT PROCESS FILENAME: that instructs a given client to start executing all the commands included in another script file named "filename", assumed to be locally available.

In general, all of the above commands are read by the Puppet Master from a file and passed on to the clients and servers. The EXESCRIPT command does not have to be understood by the clients since it is an instruction strictly for the puppet master. The Puppet Master is assumed to be failure-free.

MetaData Server A metadata server should be the second process to be launched in the system, immediately after the puppet master. No other process may be launched before at least one metadata server becomes available. Additional metadata servers may be launched, up to three simultaneously running, and metadata servers can be either orderly shutdown or can fail by not responding to requests.

Data Server When a Data Server or a client becomes active, the first operation it performs it is to notify the metadata servers. It is assumed that at least one data server is active before clients are started.

The first action of a data server is to register itself with the metadata server. It will then wait until it receives confirmation of registration. From this point on, the data server will start to receive read and write requests to provide or update content of the files stored locally.

No real failures happen in the project. Failures are simulated to mimic a *fail-stop* behavior (i.e., the failure can be reliably detected). When a metadata server (or data server) is instructed to fail it will only reply to requests from the puppet master. All requests from clients and other servers are refused and an exception is returned, indicating that the process is "failed".

Clients Clients execute sessions of reading and writing files as typical file-based applications. Clients never fail. To facilitate the debug of the program, the operation of each client is completely controlled remotely by the puppet master. Therefore, clients are in fact simple interpreters that execute commands sent by the puppet master. Each client executes as a state-machine. The puppet master and clients maintain an array of exactly 10 file-registers for metadata and that hold references to files being used, and 10 byte-array-registers for the file contents being manipulated.

7 Optional Parts

The students must first implement a version of the system that works under the assumption that the system only stores the most recent version of each file. Also, no failures of the data servers should be considered in the base implementation.

7.1 Metadata Server Fault Tolerance

Failures and recoveries of 1 or 2 replicas of metadata servers should be considered. There are 3 replicas of the metadata server. As already mentioned, failures can be reliably detected. Therefore, the metadata server should be in operation, as long as one replica is alive.

7.2 Load Balancing and Migration Strategies

In the basic system, the metadata servers may keep a record of which data servers store each file. Therefore, they can easily guarantee that each data server stores approximately the same number of files. However, some files may be much more used than others, which may result in an uneven load distribution among data servers. Also some files can have a different read/write ratio, i.e., some files can be frequently written while others are only used for reading.

In this extension, the students are encouraged to design some strategies to offer better load-balancing, by implementing a scheme to migrate data from one data server to the other, or even by changing the original read and write quorums defined for a given file. To avoid inconsistencies with the information maitained by clients, this mechanism should only make changes to files that are not in use by any client.

8 Papers

Students should begin the project by writing a paper (max. 4 pages) describing the architecture of the solution (software components and algorithms). In this paper, students should follow the typical approach of a technical paper, first describing the problem they are going to solve, the proposed solutions, and the relative advantages of each solution. The paper should include an explanation of the algorithms used and justifications for the design decisions.

The described algorithms should include at least: 1) handling entry and exit of metadata servers; 2) handling entry (and exit) of data servers and their enrolment with the metadata servers; 3) data placement algorithms regarding the selection of data servers to hold the contents of files, 4) coordination of read and write operations performed with the data servers.

The project's final report should be an extension of the paper that was previously submitted. The final report (max. 6 pages) should be as detailed as possible and include some qualitative and quantitative evaluation of the implementation. The quantitative evaluation should be based on reference traces that will be provided at the project's web site, and focus, at least, on the following metrics:

- Number of exchanged messages and corresponding size, regarding all the relevant algorithms, and how they influence both latency and the possible limits to the system throughput during heavy usage and concurrent attempts of manipulating files.

- Relative balance of the load imposed to the data servers, regarding the files whose replicas they are holding and their characteristics, namely size and operations performed on them.

This should also motivate a brief discussion on the overall quality of the protocols developed.

The papers should be written using LaTeX. A template of the paper format will be provided to the students.

The papers will be reviewed anonymously by a group of fellow students and teachers. In due time, we will post the address of the website where the papers should be submitted.

The comments to each paper will be provided to the authors. Students should incorporate these comments in their final solution.

9 Checkpoint and Final Submission

In the evaluation process, an intermediate step named *project checkpoint* has been scheduled. In 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 evaluate 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 by the checkpoint time. In contrast to the final evaluation, by the checkpoint only the functionality of the project will be evaluated and not the quality of the solution.

Therefore, for the checkpoint, students should implement the entire base system, but not any of the optional features. After the checkpoint, the students will have time to perform the experimental evaluation and to fix any bugs detected during the checkpoint.

The final submission should include the following items:

- Source code (in electronic format);
- Final report (max. 6 pages).

The project must run in the Lab's PCs for the final demonstration.

10 Relevant Dates

- March 8th - Electronic submission of the papers;
- March 17th - Electronic submission of the reviews;
- April 12th - Electronic submission of the checkpoint code;
- April 15th to April 19th - Checkpoint evaluation;
- May 10th - Final submission.

11 Grading

A perfect project without any of the optional parts will receive 16 points out of 20. The optional parts are worth 4 additional points for a total of 20 points.

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 (35% 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

The final grade of the practical components of PADI is computed as follows:

- Architecture report: 5%
- Review: 5%
- Implementation, final report and discussion: 35%

12 Cooperation among Groups

Student group elements may read the preliminary papers of other groups (in addition to the paper reviews already scheduled). You are also free to discuss alternative solutions to the problem. Furthermore, students are encouraged to help one another.

However, 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, both groups will fail the course.

13 "Época especial"

Students being evaluated on "época especial" will be required to do a different project and an exam. The project will be announced on July 8th, must be delivered July 13th, and will be discussed on July 15th. The weights are as follows: exam 65%, project 35%.