PADIMapNoReduce

Plataformas para Aplicações Distribuídas na Internet
Middleware for Distributed Internet Applications
Project - 2014-15 (IST/PADI): MEIC-A / MEIC-T / MERC / EMDC

March 7, 2015

Abstract

The PADI project aims at implementing a simplified (and therefore, not complete), in-memory implementation of the Map Reduce programming model.

1 Introduction

The goal of this project is to design and implement **PADIMapNoReduce**, a simplified implementation of the MapReduce middleware and programming model. MapReduce was introduced by Google in 2004 [1] and is currently one of the most popular approaches for large scale data analytics - also thanks to the availability of high quality open-source implementations (e.g., Hadoop[1]).

When using the MapReduce paradigm, the computation takes a set of input key/value pairs, and produces a set of output key/value pairs. MapReduce users express the computation as two functions: Map and Reduce. For simplicity, in this project students will only implement the Map part of MapReduce. The Map function (different for each application), is written by the user and takes an input set of key/value pairs and produces a set of key/value pairs. In the case of **PADIMapNoReduce**, the input key/value pairs are extracted from input files. The keys are the numbers of the line of the file being read and the values are the content of those lines.

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of *splits* of size S. The input splits can be processed in parallel by different machines, named *workers*. The system should ensure that for each job submitted, all the input data is processed. Furthermore, the system should strive to ensure good performance my monitoring a job's progress, detecting faulty or slow machines and rescheduling their tasks on idle machines.

In the original MapReduce implementation there is a centralised component, called the *job tracker*, that is in charge of supervising the progress of the job. In **PADIMapNoReduce**, the job tracker functionalities are implemented, in a distributed manner, by the workers, that cooperate to provide the tracker's functionalities. It is up to the students to decide how to distribute the tracker's tasks among the workers. The purpose of this exercise is to allow the students to get a better grasp on the advantages and limitations of distribution.

[1]`http://hadoop.apache.org/`

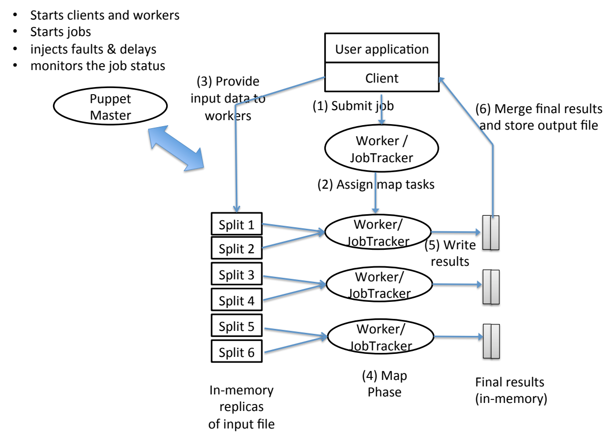

Figure 1: **PADIMapNoReduce** Architecture.

.

2 Architecture

Figure 1 provides an overview of the execution of a PADIMapNoReduce job and of the system's architecture. The system comprises three types of roles, which may be physically distributed on different machines:

- user level application, which submits map jobs to the system. For simplicity we assume that there is only one user level application that submits its jobs synchronously and that the node over which it is running never crashes.
- client, which is physically co-located with the user-level application and that exposes to user-level applications a method to submit jobs in the system.

  In addition, clients export two services that are used to interact with the remaining nodes in **PADIMapNoReduce** platform: one that provides fragments of input file (called splits) to the workers, and one to receive the output data resulting from the processing of a split.
- worker processes, i.e. processes that are in charge of executing a job. The set of workers is denoted as W.
- puppet masters, which serves only for testing purposes, and allows for injecting failures/delays in the system, as well as for creating instances of the worker processes and for submitting jobs. There should be one puppet master per machine being used in the **PADIMapNoReduce** system.

Next we describe each phase of execution of a job:

1. The user-level application submits the job to the **PADIMapNoReduce** system, specifying the implementation of the Map function, the name of the input file for the computation, and the number of splits S in which the input file should be partitioned.
2. The **PADIMapNoReduce** worker(s) in charge of job tracking decides how to divide the job's input file into S splits of equal size. Note that, typically, $S > |W|$. The input file is assumed to be accessible only at the client.

3. The system assigns the map tasks to the worker nodes. The assignment should be performed in order to maximize data locality and minimize data transfer over the network. Also, the assignment of tasks to workers may be done in an incremental fashion, e.g. new maps tasks are assigned to workers only when the previous ones have been completed.
4. A worker who is assigned a map task asks the client for the contents of the corresponding input split. The workers ask the client for his input data by providing the two positions in the input file where his split begins and ends. After the client returns the split to the requesting worker, the worker then parses the split one line at a time, and each line is passed to the user-defined Map function. Each invocation of the Map function returns a set of key-value pairs. The set of key-value pairs produced during the various executions of the map function are grouped by key and are stored in memory.
5. When the map phase is completed (i.e. all lines of the input file have been processed), workers nodes call the result submission service at the client to submit the results of processing their split(s).
6. When the job completes, the user level application is notified by its local client.

3 Application API and Clients

In order to use the computational services offered by the **PADIMapNoReduce** platform, user level applications have to first call the INIT(STRING ENTRYURL) method, which will cause the activation of a client process on the local machine. EntryURL is the URL of a worker node to which the client can connect to submit jobs.

User level applications submit jobs by invoking the SUBMIT method of their local client, which takes as input the following parameters:

- The path to an *input file*, stored on the local file system, which will be used as input for the job. Note that the input file is accessible only on the local node on which the application and its client are running. The input file, instead, is not accessible by worker nodes, which need to retrieve the input data by the client.
- The number S of splits in which the input file shall be partitioned.
- The path of an *output directory* on the local file system, which will be used to store the files containing the output of the processing of the input splits, and which shall be named"1.out", "2.out", ..., "S.out".
- A class implementing the IMap interface, which exposes a single method, i.e.:
  `Set<String key, String Value> Map(String key, String value)`
  and that returns a set of key-value pairs for each input key-value pair.

The SUBMIT method is executed synchronously, i.e. it returns to the user level application only when the MapReduce job is fully executed.

As already mentioned, clients expose two services, which serve to: i) return a split of the input file to a worker node, and ii) obtain the results generated by the processing of a split from a worker node.

4 Job Trackers

As noted previously, in the original MapReduce design, there is a special node, called job tracker, responsible for scheduling and coordinating the execution of Map Reduce jobs. In

PADIMapNoReduce, this functionality should be integrated into the workers. Students should carefully design how the system management features are assigned to the workers.

The functionalities included in job coordination are:

1. deciding how to split a job into worker Map tasks;
2. assigning map jobs to the worker nodes;
3. monitoring job execution and possibly notifying the client application when the execution of a submitted job is completed;
4. detecting faulty nodes, or slow nodes (also called straggler nodes) that are delaying the completion of the map tasks. In both cases tasks should be rescheduled on some idle worker node.

Job coordination behaviour cannot be implemented at the clients.

5 PuppetMaster

To simplify project testing, each machine of the system will execute a PuppetMaster process. PuppetMasters will be the first set of process to be activated in the **PADIMapNoReduce** platform. The activation of the PuppetMasters will be performed manually, for simplicity, although they could be configured to be executed as daemons.

PuppetMasters provide the following functionalities:

1. They expose a service at an URL, called *PuppetMasterURL*, that creates worker processes on the local machine. For simplicity, we assume that all the PuppetMaster know the URLs of the entire set of PuppetMasters. This information can be provided, for instance, via configuration file or command line.
2. They expose a simple GUI interface supporting the execution of sequences of command, called scripts. The GUI shall allow to load a script, execute it step-by-step and without interruptions. The GUI should also permit the submission of individual text commands, e.g. via a text box with a "Submit" button. Additionally, PuppetMasters should provide a worker creation service that can be called by other PuppetMaster to create workers. Students can assume that the GUI of a single puppet master will be used during a test.

The puppet master commands allow users to:

1. start worker processes;
2. start an application (and its local client) and submit jobs;
3. show the execution status of submitted jobs;
4. inject delays and failures in various stages of the execution phase of a job and across different components of the system.

More in detail, (at least) the following commands must be supported:

- `WORKER <ID> <PUPPETMASTER-URL> <SERVICE-URL> <ENTRY-URL>`: Contacts the PuppetMaster at the `PUPPETMASTER-URL` to creates a worker process with an identifier `<ID>` that exposes its services at `<SERVICE-URL>`. If an `<ENTRY-URL>` is provided, the new worker should notify the set of existing workers that it has started by calling the worker listening at `<ENTRY-URL>`. Since this command can be used to create local or remote workers, it will be simpler to implement it as a call to the local (or remote) PuppetMaster's job creation service.

- `SUBMIT <ENTRY-URL> <FILE> <OUTPUT> <S> <MAP>`: Creates an application on the local node. The application submits a job to the **PADIMapNoReduce** platform by system by contacting the worker at `<ENTRY-URL>`. The job is defined by the following parameters:
  - `<FILE>` is the path to the *input file*. The file will be subdivided into `<S>` splits across the machines in W.
  - `<OUTPUT>` is the path to an *output directory* on the local filesystem of the application, which will store one output file for each split of the input file name "S1.out", "S2.out", ..., "S.out".
  - `<S>`, i.e. the number of splits of the input file, which corresponds to the total number of worker tasks to be executed.
  - The name of the class implementing the IMap interface.
- `WAIT <SECS>`: Makes the PuppetMaster stop the execution of commands of the script for `<SECS>` seconds.
- `STATUS`: Makes all workers and job trackers of the **PADIMapNoReduce** system print their current status. The report shall allow to determine the state of progress of each map task at every node of the platform, as well as the current phase of execution of the job (e.g., transfer of the input/output data, computing, etc.). The status command should present brief information about the state of the system (who is present, who is in charge of coordination, which nodes are presumed failed). Status information can be printed on each nodes' console and does not need to be centralized at the PuppetMaster.
- `SLOWW <ID> <delay-in-seconds>`: Injects the specified delay in the worker processes with the `<ID>` identifier.
- `FREEZEW <ID>`: Disables the communication of a worker and pauses its map computation in order to simulate the worker's failure.
- `UNFREEZEW <ID>`: Undoes the effects of a previous `FREEZEW` command.
- `FREEZEC <ID>`: Disables the communication of the job tracker aspect of a worker node in order to simulate its failures.
- `UNFREEZEC <ID>`: Undoes the effects of a previous `FREEZEC` command.

Any line in a PuppetMaster script starting with a "%" sign should be ignored.

6 Advanced Features

The project includes two advanced parts related to the implementation of functionalities for the management of faults affecting various components of the system. Students shall implement these advanced features only having fully tested the correctness and efficiency of a base (non fault-tolerant) implementation. We describe the advanced features in the following subsections.

6.1 Fault tolerant implementation of the job tracking

The job tracker is a critical component of the MapReduce framework and represents a single point of failure of the system. Students shall design and implement a replication mechanism capable of ensuring the high availability and correctness of the job tracking functionality despite the occurrence of failures at the nodes responsible for it.

6.2 Masking failures of worker processes

In the **PADIMapNoReduce** framework the input data of a MapReduce job is maintained in-memory at the worker nodes. In presence of failures of the worker nodes, hence, data could be lost and it may be necessary to incur non-negligible costs, e.g. to re-execute the mapping phase or to fetch again data from the input file at the client. Students shall design and implement data replication aimed to mask the failure of a single mapper. The design shall keep into account also of the overheads introduced by the replication mechanisms during the normal execution of the platform, i.e. in absence of failure, and identify adequate trade-offs between the advantages in terms of failure resiliency and additional costs incurred by the designed data replication mechanisms.

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 the problem they are going to solve, the proposed solutions, and the relative advantages of each solution. The report should include an explanation of the algorithms used and justifications for the design decisions. The project's final report should also 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 on the following metrics:

- Job execution time, both in failure free scenarios and in presence of injected faults;
- Number of messages and size of messages exchanged among nodes;
- Size (in bytes) and number of the final key-value pairs emitted on average.

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

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 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 by the checkpoint time. 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.

Therefore, for the checkpoint, students should implement the entire base system, but not any of the advanced 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 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

- April 10^{th} - Electronic submission of the checkpoint code;

- April 13th to April 17th - Checkpoint evaluation;
- May 15th - Electronic submission of the final code.
- May 18th - Electronic submission of the final report.

10 Grading

A perfect project without any of the advanced features will receive 15 points out of 20. The advanced features are worth 5 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 (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, both groups 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 project will be announced on July 13th, must be delivered July 20th, and will be discussed on July 22th.

The weight for the project is of 45% (as for the "Época normal"), if the student has given the paper presentation (which will have weight 15%) during the semester (and the exam will account for the remaining 40% of the grade).

If the student has **not** given the paper presentation during the semester, the weight of the project will be 60% (and the exam will account for 40% of the grade).

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.