PADIMapNoReduce

João Santos Filipe Guerreiro Daniel Castro
Instituto Superior Técnico
MEIC-T
70495, 73940, 73997
joaombsan@gmail.com, filipe.m.guerreiro@gmail.com, daniel_me@live.com.pt

Abstract

In this paper we present PADIMapNoReduce, a simplified in-memory implementation of the MapReduce model, and we describe what algorithms and mechanisms we used to implement it.

Unlike MapReduce, PADIMapNoReduce features distributed tracking techniques to account for all the nodes in the cluster.

Our implementation of PADIMapNoReduce successfully incorporates all the features desired from the base system, as well as the mechanisms for fault-tolerant workers and trackers.

1. Introduction

There is much demand for handling large amount of data quickly. This usually leads to developers building a specific application specifically to work in large clusters of machines, which adds an unnecessary amount of complexity (and bugs) and time to an application.

MapReduce is a middleware and programming model designed work in these environments and allow the developers to worry less on the infrastructure underneath. MapReduce demands only that a set of key/value pairs are taken as input. Then an output of key/value pairs is produced when processed through two functions: Map and Reduce. These functions are defined by the developer as well.

PADIMapNoReduce is a simplification of MapReduce where only the Map function is used. The system has a *client* that provides the input, and a set of *workers* in a cluster that take the input and use the Map function to produce the output.

The input will be split into several parts and divided among the workers. The task of coordinating how this division is made and tracking the progress of the job is the task of the *tracker*, which can be performed by any worker in the cluster.

The remainder of this article describes our solution to this problem and its architecture.

2. Architecture

2.1. Component Overview

In this system, it is important to identify the following components:

- **User-level application**, is responsible for submitting jobs to the system.
- **Client**, which is responsible for communicating with the workers, either for submitting jobs and receiving the results. This is also co-located with the User-level application.
- **Worker**, which receives the input from the client, applies a Map function and delivers back the results.
- **Tracker**, is a module co-located with the Worker component and is responsible for, given a job, calculate the list of chunks the job will be splitten into, and assigning it to the other workers in the network.
- There is also a final component, called the **PuppetMaster** which is used for initialization and testing, and for providing a GUI for the user to send commands. This component is co-located with clients and workers.

2.2. Execution Overview

The user that wishes to use the PADIMapNoReduce system must provide an **Input file** to be processed, a

Map function that will be used to process it, and a **Split** number, that defines the number of chunks the input file will be divided into.

These parameters are sent to one of the workers in the system, which will do the following: calculate how many splits each worker in the system will have to do, and inform each of them of their assignment.

Each of these workers will then ask the client for the part of the input file they are in charge of, and apply the Map function. Once they are done, they will send back the results to the client.

3. Implementation

In this section we'll present our PADIMapNoReduce implementation of each component.

3.1. PuppetMaster

The PuppetMaster is the first component of the system to be instantiated, and must be done so manually. When a PuppetMaster instance is launched, a client can also be instantiated alongside the PuppetMaster by giving the command to launch the GUI (in our case, the argument –Dgui).

The PuppetMaster then exposes the following, more important, commands:

- WORKER <ID> <PUPPETMASTER-URL> <SERVICE-URL> <ENTRY-POINT>: **initiates a new worker**. They can be created in the same process as the PuppetMaster or in another machine/process where another PuppetMaster exists (according to the provided URL).
- SUBMIT <ENTRY-WORKER-URL> <INPUT-FILE> <OUTPUT-FOLDER> <NUMBER-OF-SPLITS> <MAPPER-CLASS-NAME> <DLL-OF-MAPPER-IMPLEMENTATION> **Submit a job to the system**. This works only if the PuppetMaster previously launched a Client (by default each PuppetMaster launches a Client so the command will work in each PuppetMaster).
- the other commands **introduce faults and delays in the workers**. They are SLOWW <ID> <TIME-IN-SEC> (slows target worker the given amount of time), FREEZEW <ID> / UNFREEZEW <ID> (pauses/resumes target worker thread, tracker stays up) and FREEZEC <ID> / UNFREEZEC <ID> (pauses/resumes target worker tracker, and communications with other workers / client).

3.2. Client

The client is used exclusively for submitting a job, which requires the following parameters:

- **Input File path**, the path to the file that the user wants to have processed by the workers.
- **Map function**, a function that takes a set of key/value pairs and returns another set of key/value pairs.
- **Number of splits**, the number of chunks the input file will be divided into.
- **Output directory**, the filepath in the local client directory space where the output will be written.
- **Address of one worker/tracker**, the worker/tracker to which the job will be submitted to.

After submitting a job, the client will receive requests for parts of the input file from each of the workers.

The client can only submit one job to the system at the time, and when it does so, it will wait until all the splits have been returned.

3.3. Worker

The Worker's job in the system is to apply the Mapping function to each line (key-value pair) of each split that it receives.

Failures notwithstanding, a worker's life will pass through a few stages, namely:

- **IDLE**, the initial state while waiting for a submit request. When a request comes in from one of the trackers, it receives the Mapping function that it will use for processing the job, as well as the client address so he can ask for the input file.
- **ASK_INPUT**, after receiving a submit request, it will send the client a request for each of the splits it was assigned to, and compile it into a list for later processing.
- **COMPUTE**, it is at this stage that the real work of the system is done, for each key-value pair of each split that is obtained, a Mapping function obtained previously is applied and stored in-memory for later.
- **SEND_RESULTS**, when all the map jobs have completed, the worker will notify the master tracker that it has completed the job, contact the client and send the splits one by one. Afterwards, it will return to the IDLE state.

3.4. Tracker

The Tracker is a component that is attached to each Worker in the system.

As previously mentioned, the tracker is the component responsible for **assigning the job splits to each worker**. It does this when receiving a submit request from the client. It simply calculates the size of each split by diving the size of the input by the number of splits the client wants; then takes the number of splits again and divides by the number of workers available to obtain the job assignment for each worker. It then sends each of them, their job assignment as well as the Map function and the client information.

In order to do it's job correctly, it is responsible for **knowing every worker in the system**, storing them in a list. This list is updated when a new Worker is added into the network or removed when suspected to have crashed.

When a worker is added to the network, it receives the list of current workers in the network, and sends a message to all current workers of it's arrival.

For every worker/tracker that is added to the network, one other tracker is initially informed of its arrival. This tracker that is initially informed will have the job of checking if the new worker/tracker is alive. This is done by setting a timer, and then waiting to receive lifeproofs periodically from the new node.

Another job of the tracker is to check for straggler workers (workers that take too long to finish a job). Each of the job assignments each worker receives is the same size, so when one of the workers notify the "master" tracker (that received the job request from the client), the "master" tracker will set a timeout value for a few seconds, waiting for all the other workers to finish. After the timer goes off, it search the assignments that it did not receive notifications for, and reassigns it to another worker that is free.

3.5. Fault-tolerance

Since this system is designed to help process very large amounts of data using hundreds or thousands of machines, the system must tolerate machine failures gracefully.

3.5.1 Tracker Fault-tolerance

Each worker's tracker is responsible for sending lifeproofs to the next worker and track if is accessible (using a timeout system). For example, 3 workers were created, worker 1 (W1) does not have an entry URL, worker 2 (W2) has W1 as entry URL and worker 3 (W3) has W2 as entry URL. So W2 must send lifproofs to W1 and wait for an acknowledgment (ack), on W1 side, it must wait from W2 lifeproofs for a fixed amount of time before assume that W2 crashed.

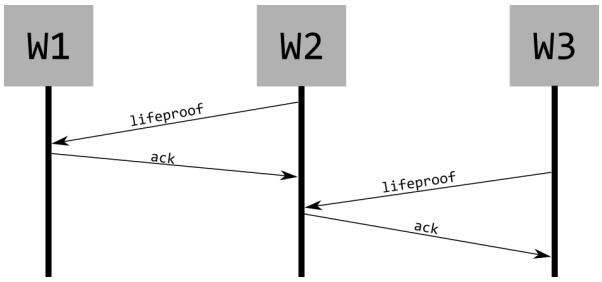

Figure 1. Illustration of the tracker activity sending lifeproofs.

The figure 2 shows the situation where first W2 crashes and then, after system recovers from that crash, W1 crashes. W3 sends a lifeproof to W2, but because W2 has crashed, the .Net Framework throws a RemotingException due to the W2 tracker being down (a SocketException may be thrown as well due to the TCP channel going down during the communication). Also W1 will be aware that W2 is not working properly because it didn't receive a lifeproof from W2 in a while. W3 has information that W1 is on top of W2 (the lifeproof messages have that information) and sends a lifeproof to W1, then W1 is able to track W3 activity.

If W2 recovers from the crash it would send a lifeproof to W1 and W1 would start a timer to track W2 (and will continue to track W3).

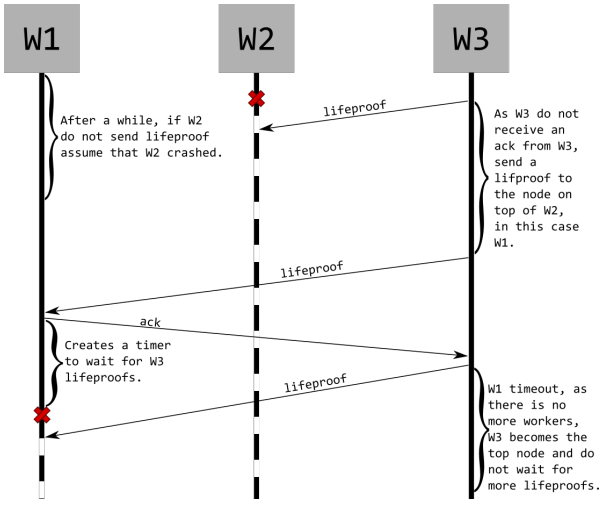

Figure 2. Illustration of a crash.

3.5.2 Worker Fault-tolerance

If a worker in the system fails, it will force another worker to start the job entirely from zero. This requires the new worker to requesting again the input from the client, applying the Map job to it, and return it. This can easily double the total time the system takes to finish a job. If we have very large files in the system, this wait time is likely non-negligible.

We would like to reduce the first two additional times: asking the client for input, and processing again all the jobs done before the worker crashed.

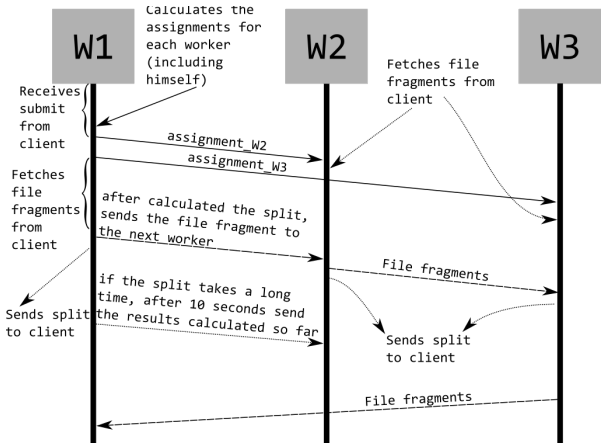

Figure 3. Illustration of workers processing the client's submit.

Our solution to this problem comes from:

- For the set of workers in the network, we have so that each worker (W) from the set has one other worker (W'), belonging to the same set, to which redundant information is sent.
- To ensure that worker W', in case W fails, does not need to request the input file again, W sends the input file when he receives it, and W' stores it in memory.
- For the processed job, the system is similar, W sends periodically (10 in 10 seconds) the lines that he has processed so far.
- When the worker W fails, the master tracker (the tracker to which the job was submitted to), reassigns the job to worker W', which checks for the information it has stored before continuing the job.

4. Evaluation

In this section, we measure the performance of our implementation using a 5 MB text of Sir Arthur Conan Doyle's work, the CharCountMapper function and using up to 8 workers and 20 splits, on the same machine. The workers will be running in separate processes.

4.1. Execution time

To measure the execution time of our implementation, we set a stopwatch on the client to measure the time it takes from the moment a submit is issued, until all the results are received.

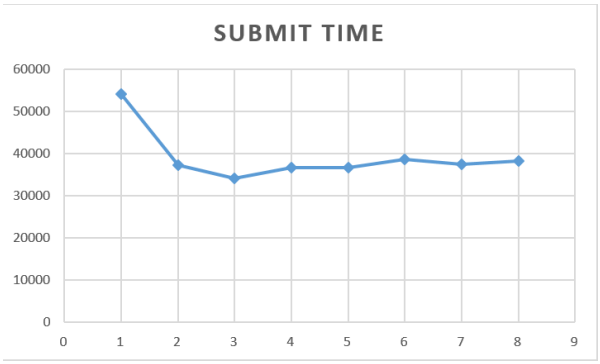

Figure 4. On the Y axis we have the time in milliseconds that the submit job took, while in the X axis we have the number of workers in the system.

As you can expect, the time it takes is largest when we have a single worker. And, as you can see, we have a 40% improvement in performance when we add a second worker, but a much smaller increase when we add a third one. Adding more workers to do the job does not improve job performance. We give two reasons for this: the first is, the client will act as a bottleneck to the system, since in our system, the client will have one thread to write the results of each worker. The second lies in the additional network communication that takes place from the fault-tolerance mechanisms.

4.2. Execution time in presence of faults

Next, we checked what would happen to the total time if one worker failed 5 seconds after he would begin his work. We present below the graph and compare it with the values obtained above.

The immediate thing to notice is the big spike where we have two workers and one of them crashes. One

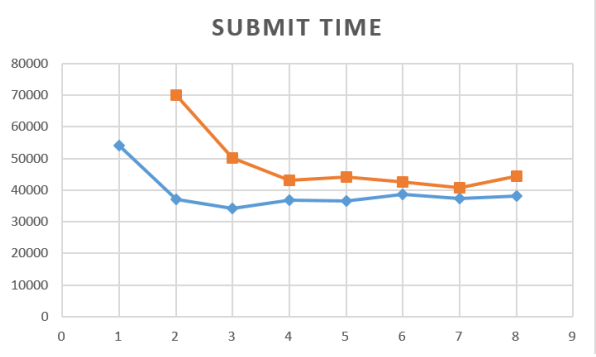

Figure 5. Here we have the same graph as before, but the orange line shows the time with 1 crash.

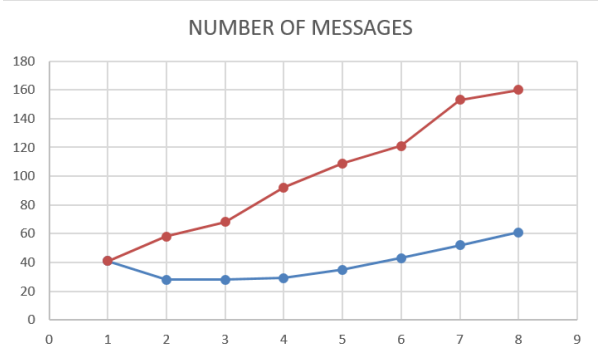

Figure 6. On the Y axis we have the number of messages exchanged, while in the X axis we have the number of workers in the system.

worker will finish in around 40 seconds, however, if we account for the extra communication delays and timeout delays (the delay one worker will take to detect another is faulty), the result is not so surprising.
And as expected, as the number of workers increases, the impact of its removal decreases.

4.3. Number of messages between nodes

We were curious to determine the load on the network as the number of workers increase. For this analysis, we used the same conditions as before, namely, 5 MB file and 20 splits.

We tried to account for all types of messages in the system, such as lifeproofs, assignments by the master tracker, requests for splits, delivery of splits, etc. We started the counter as soon as a worker enters the network until delivering all the results to the client to account for the messages exchanges when new workers are added.

The line in red is the TOTAL messages exchanged, while the BLUE line is the messages sent by the Master Tracker.

When we have 1 worker, it exchanges messages with the client only, 20 messages account for asking for the input file, while another 20 account for sending the results. When we have 2 workers, the number of messages sent by the Master Tracker lowers because it has to ask and send fewer splits (10), but it now has to send additional messages to track the list of workers available and assign jobs to each. As the number of workers increases, the number of total messages and messages sent by the Master Tracker increases somewhat linearly due to the linear nature of lifeproofs, worker additions and job assignments.

5. Conclusions

In this article, we presented a solution for PADIMapNoReduce and we described our implementation of the advanced features. We feel there is still room for improving, debugging and refactoring, but in the end, we think we achieved a satisfactory implementation of the proposed project.