Testing Fault Tolerance in Map Reduce

Adriana Tufa, SCPD

Introduction

The goal of the project is the analyse and assess how MapReduce framework behaves in unpredictable situations. What happens with nodes, tasks and data when different types of failures and exceptions appear and what is the performance penalty.

The first thing to observe is that Hadoop 2.x uses a different architecture from Hadoop 1.x, which was more prone to unwanted failures. Hadoop 2.x with MapReduce have three layers of abstraction to form a more robust and scalable platform. At the bottom lies HDFS which implements its own fault tolerance mechanisms. Above it there is YARN, which is a resource manager system that orchestrates the allocation and monitoring of different jobs on the available nodes. YARN offers the possibility to make the cluster usable by MapReduce and other frameworks at the same time (which was not possible in the first version of Hadoop). It is platform agnostic as it exposes an interface for applications to request their needed resources.

Related Work

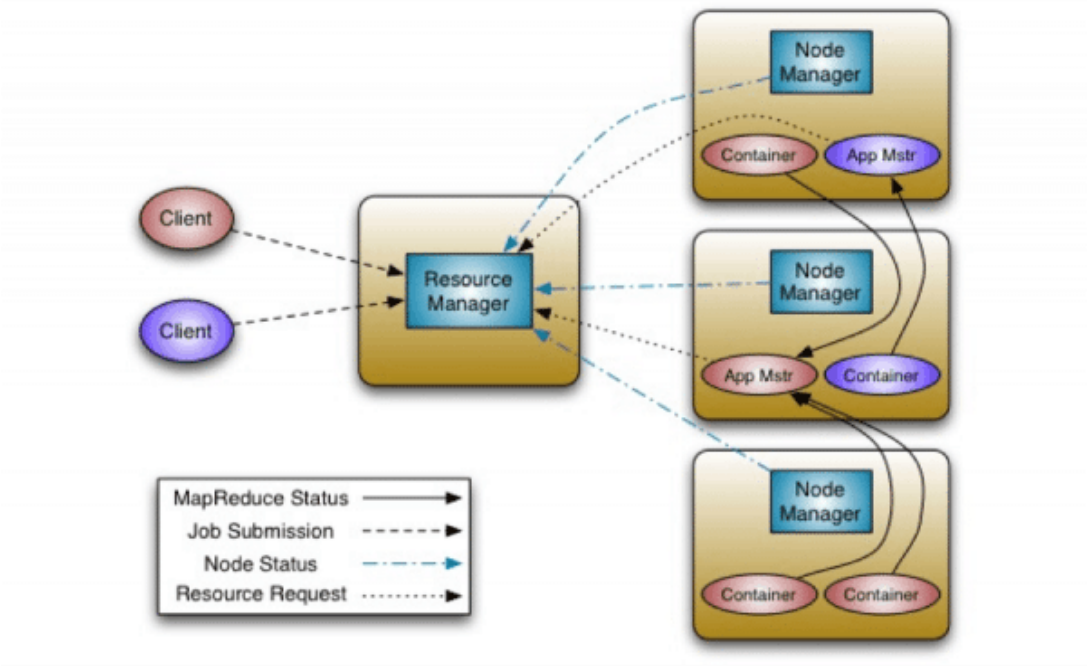
There isn't a standardized or a correct way of testing the fault tolerance of a distributed system, only common scenarios which can be tested. Similar work has been done in creating frameworks that allow to define the test cases for a MapReduce application in a programmatic and straightforward way. Then, using reflection, the frameworks would get hold of the actual workers and simulate their creation or failure.[1][2] The later uses Petri networks to create relevant test cases. But these are only used for the MapReduce layer and do not dig into what happens with YARN.

Architecture Design

The main components of the architecture are:

- a global ResourceManager - it received requests from Application Managers to allocate containers with specified resources (CPU, memory). It schedules the container and finds an available node for each request.
- a per-application ApplicationMaster - each application running on top of YARN has an Application Master which handles and communicates with their workers.

- a per-node slave NodeManager - Each physical machine has a Node Manager which monitors the containers running on that node.
- a per-application Container running on a NodeManager - this is a worker running a specific task, as specified by the ApplicationManager.

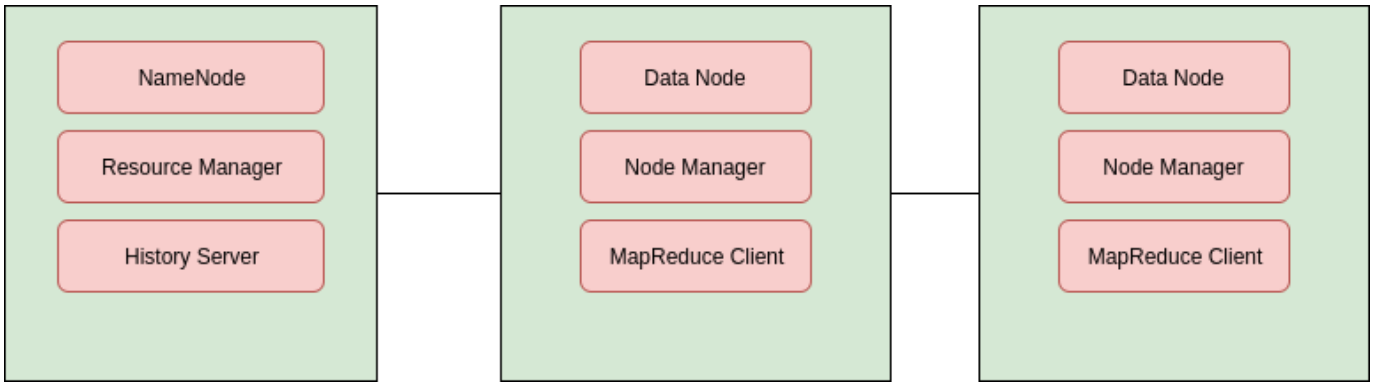


Essentially an application can ask for specific resource requests via the ApplicationMaster to satisfy its resource needs. The Scheduler responds to a resource request by granting a container, which satisfies the requirements laid out by the ApplicationMaster in the initial ResourceRequest.

Given this architecture, the aim of the project is to test different scenarios which would make use of the fault tolerance property of the systems and observe how it affects the overall performance.

Implementation Details

For testing the fault tolerance of map reduce we have to have in place at least 3 systems: Hadoop, Yarn and Map Reduce framework. The experiment was conducted on a local machine, with limited resources (16 GB RAM, 500GB HDD, i5 CPU with 4 cores). Taking this into account, we chose to make a virtual cluster with docker containers. Alongside these 3 systems, we used Apache Ambari for an easier management of the cluster and service monitoring. The final system looks like the following:



Each Node Manager can leverage maximum 4GB of memory and 2 cores and the name nodes/ data nodes can use all the available memory on disk. Also, the Resource Manager can allocate containers of a maximum of 1.5 GB.

For the actual testing we first ran a map reduce application for generating random data (10GB for each node). Then, we tested fault tolerance on an application which sorts the data, so it is mostly compute intensive. Sorting all the generated files took too much time to be a viable solution, so we reduce the dimension to only 40% of the generated data.

Next, we constructed several test cases:

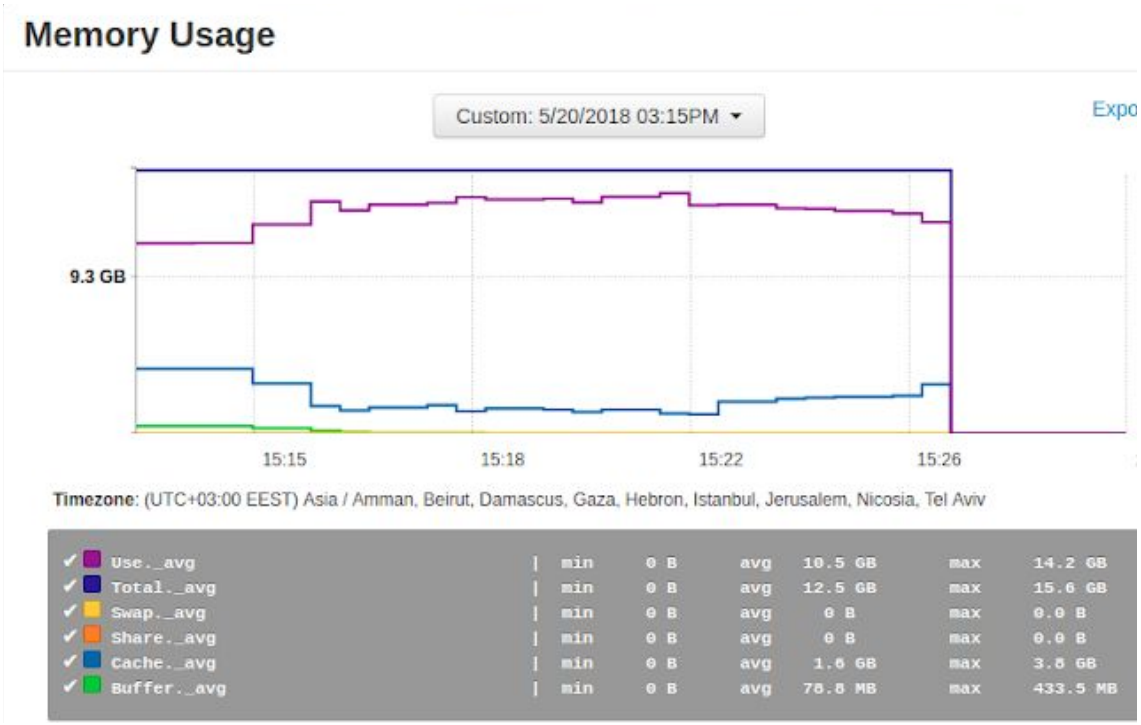
- Normal run, with no failures
- A run where a container fails
- A run where multiple containers are killed repeatedly
- A run where Application Master fails

We measured the running times and other metrics and asses the impact these scenarios have on our modest cluster.

Experimental Evaluation

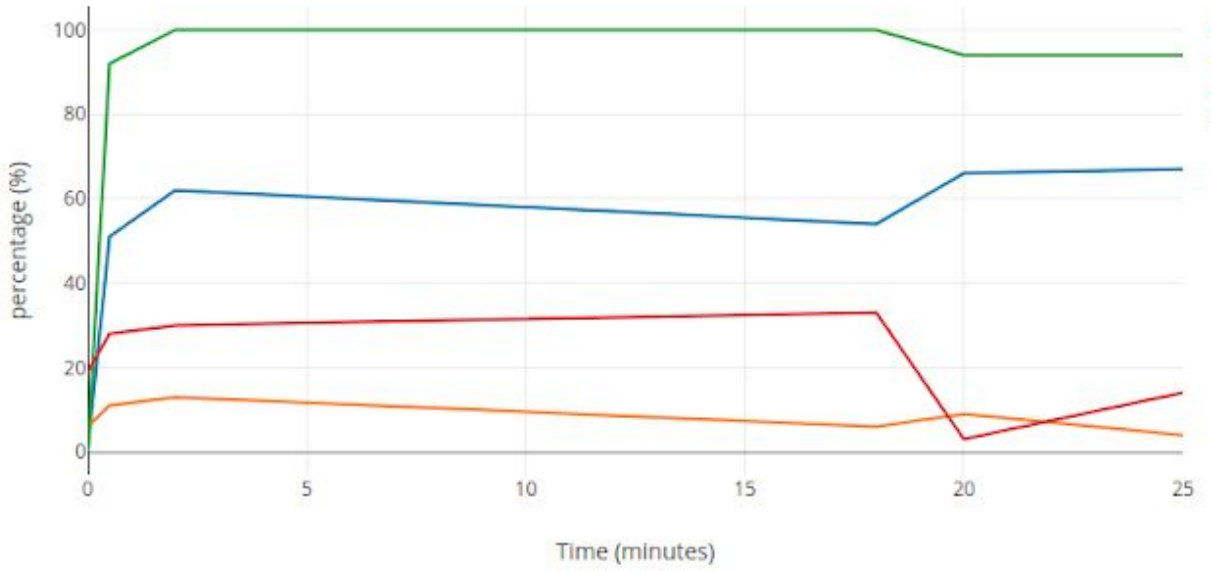
1. **Simple run**

In the normal run of the application we measured different resources in order to better understand how they are used.



The used memory started for 9.8 GB and reached 14.2 GB. This represents only the memory used for running the map reduce job.

We can see how the cluster gets increasingly loaded. It reaches it peak during the time it still has map tasks, because it allocates all the available resources.



NameNodeCPU, NameNode Heap, Yarn Memory, Resource Manager Heap

The above plot illustrates more information about the resource usage during the run. We can see that the NameNode CPU raises to ~60% and stays constant during the run but the heap changes slightly. It is more solicited in the beginning, when the mappers need to read the data. The Yarn memory is used up almost instantly. It allocates the entire space to all the containers. For the Resource Manager heap we can see that it is more used in the first ¾ of the time, as it is occupied with allocating space and scheduling mappers, which are in much greater number than the reducers.

The running time was 735s and the job created 40 mappers and 3 reducers.

2. **Kill one running container**

Giving rest commands to the apache server does not take immediate effects on the nodes as it wait for them to finish their job and then deletes/kills certain components. So the only sure way to kill a container is to actually do it directly from one of the nodes. So we launched the mapreduce job on the second node and on the third node we failed a container with mapred commands. We chose a container running a reducer because they take considerably longer than the mappers and have a greater impact.

We killed the container when the reduce part was at 30% and we got a penalisation of 140s. The reduces started all over again and on the other node.

3. **Kill running container repeatedly**

We also played around with killing reducers continuously in order to see how the application reacts. We made a script that checks the reducers every 2 seconds and fails the tasks. It

seems that after 10 repeated failed tasks, the application itself fails and does not try to start any other reducer.

We started killing reducers when the reduce part was at 14% and the application stopped at 599s.

4. **Kill Application Master & Node Manager**

There was no clear way of how to get hold of the application master itself and kill only that specific container so we took a different approach. We killed the JVM on the node that the Application Master was running, thus testing the node manager fail as well. We observed that the mapreduce job still continues and creates a new Application Master. Moreover, the work that has been done until that point was preserved and the application resumed from almost the same state. This means that there is a checkpointing system in place for mapreduce which takes care of not doing the same computation multiple times.
One thing to note is the Node Manager does not recover and it needs to be restarted manually.

The job took considerably more time, 1402 seconds, almost double the running time from the simple test case.

Conclusions

To sum up, this paper proposed a way of testing fault tolerance in Map Reduce and implicitly Yarn on a local machine. It tested Map Reduce jobs started from a sorting application and observed the performance impacts of each type of failure. We tested with container and components failures. The test can, of course, be extrapolated to an actual distributed system, with more concludent examples for real life scenarios.

References

[1] Bunjamin Memishi, Shadi Ibrahim, Marıa S. Perez and Gabriel Antoniu, Fault Tolerance in MapReduce: A Survey
[2] João Eugenio, Marynowskiab Altair, Olivo Santina Andrey, Ricardo Pimentelb, Method for testing the fault tolerance of MapReduce frameworks