



Resiliency

This lesson explains how failures of various components are handled in the MapReduce framework.

We'll cover the following ^

- Resiliency
- Task failure
- ApplicationMaster failure
- Node Manager
- Resource Manager

Resiliency

So what is so special about MapReduce? After all, the same problems can be solved on other platforms or supercomputers. The lure of MapReduce is its ability to run on cheap commodity hardware as there's a high probability of hardware and other infrastructure breaking down. In this lesson, we'll see how running a MapReduce job handles various failures.

Task failure

A map or a reduce task for a MapReduce job can fail for many reasons: bug in the user's code, bug in the JVM, insufficient space on the drive, and others. When a task fails, the JVM hosting the task reports the error message to its parent application master before exiting. The ApplicationMaster marks the task attempt as failed. It reschedules it for execution on a different node manager if possible. There is a configurable number of times that map and reduce tasks are allowed to fail before the entire job is marked as failed.

CHUNK JOB IS MARKED AS FAILED.




How are long-running tasks handled? Tasks that don't send a progress update after a configurable number of minutes are deemed failed by the ApplicationMaster and the JVM running the task is killed. We can tweak the number of minutes the ApplicationMaster waits before marking a non-responsive task as failed and even disable killing the task altogether.

In some cases, we may not want to fail the entire job if a few of its tasks fail. We can control this behavior by a configurable property that specifies what percentage of tasks should fail before marking the job as a failure. This allows a job to run to completion even if a few tasks fail.

ApplicationMaster failure

The ApplicationMaster process can also fail. The YARN Resource Manager receives periodic heartbeats from ApplicationMaster and can detect when the ApplicationMaster fails. The Resource Manager restarts the ApplicationMaster in a different container managed by the Node Manager. Similar to how there's a limit on the failures of the map and reduce tasks before a job being declared as failed, the ApplicationMaster is subject to limits on the number of times it can fail. In case of a restart on a failure, the client needs to be updated with the new location of the ApplicationMaster. The client polls the ApplicationMaster at the old address for progress and the connection times out. It then contacts the Resource Manager for the new location of the ApplicationMaster and re-establishes contact. Also, the Resource Manager, when restarting a failed ApplicationMaster can recover the state of the tasks run before the failure from the job history server so they aren't run again.

Node Manager

The Node Manager sends a periodic heartbeat to the Resource Manager.  the heartbeat isn't received by the Resource Manager within a

dead. If so, the Resource Manager restarts any failed tasks



ApplicationMasters running on the failed Node Manager on other, healthy nodes. However, map tasks completed on the failed node must be re-run. Their intermediate output resides on the local filesystem of the failed node; it can't be forwarded to the reduce tasks as the node isn't functional anymore.

Resource Manager

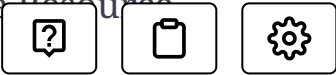
The Resource Manager, like the rest of the components is also prone to failures. However, failure of the Resource Manager is critical because if it goes down, then we cannot launch new tasks, failed tasks, or ApplicationsMasters. Usually, two Resource Managers are run together as a pair in an active/standby configuration. The active Resource Manager instance initiates and responds to all interactions. The standby Resource Manager lies dormant until the active instance experiences a failure. At that point, the standby takes over and becomes active. This aspect is known as *High Availability*. It is a characteristic of distributed services expected to keep working when sub-components fail.

The transition of the Resource Manager from standby to active is conducted by a failover controller. This controller uses *Zookeeper* leader-election to ensure only one active Resource Manager. Zookeeper is a service offering configuration storage, naming, distributed synchronization, and other group services to distributed applications.

Zookeeper keeps information about running applications. When a standby Resource Manager becomes active it reads that information to reconstruct the state before the other Resource Manager went down. The applications are restarted by the newly active Resource Manager. However, that doesn't mean that the jobs start from scratch. The built-in resiliency mechanism allows the applications to recover the completed tasks as described earlier. Information relating to Node Managers is constructed from the received heartbeats and not stored in Zookeeper.



Finally, the Node Managers and client applications handle Resource



Manager failures by connecting to the Resource Manager instances in a round-robin fashion, until the active one is found.

← Back

Putting it Together

Next →

Filesystem

☒ Mark as Completed

Report an Issue

