Providing fault tolerance and scalability of the MapReduce JobTracker using the Infinispan platform

(extended abstract of the MSc dissertation)

Strahinja Lažetić
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor João Coelho Garcia

Abstract—The Apache Hadoop MapReduce is a distributed framework for processing large amounts of data in parallel on a cluster of computers. As many large scale processing distributed systems, MapReduce uses a master-slave architecture, with the JobTracker serving as a central master component of the system and multiple slave workers called TaskTrackers. The JobTracker runs on a single physical node and represents a single point of failure of the system; when it fails the systems loses all of its running jobs and becomes unavailable to the clients. In addition, a single physical node represents a constraint to the system’s scalability.

This thesis makes a contribution to both previously explained problems. We design and implement a fully functional solution for the JobTracker fault tolerance. We create a simple and efficient solution for providing the scalability of the JobTracker. As a support for providing both solutions we use the Infinispan data grid and distributed processing framework. We perform different tests in order to evaluate our solutions and discuss the obtained results.

Keywords—Fault tolerance, Scalability, JobTracker, Infinispan, Replication, Distributed execution

I. INTRODUCTION

Nowadays, distributed systems cover a wide range of systems from the most famous World Wide Web to large scale processing frameworks [1][2]. One of the most complex challenges in the design of a large scale processing distributed system is how to provide efficient coordination and communication between the system server nodes. A quite common approach supposes the use of a master-slave architecture [3][4]. Although master servers can have diverse roles in different systems, they are usually centralized and run on a single physical node, thus representing a single point of failure. The implications of a possible crash of the master are in many cases severe. Another problem connected to a single node master server is the issue of scalability. If in some usage scenarios, the master can have a too demanding task in terms of processing and memory consumption and thus can reach its operational threshold. These two issues impose a new task on the design of a distributed system - providing high availability and scalability of the master server.

The Apache Hadoop MapReduce is a distributed framework for processing large amounts of data in parallel on a cluster of computers [1]. The main components of the framework are a single master node of the system, the JobTracker, and multiple slaves, one per cluster-node, called TaskTrackers. The JobTracker receives client jobs to be executed on the cluster, delegates the jobs’ tasks to the TaskTrackers, monitors them and restarts in case of failure. The TaskTrackers are the workers who process the jobs’ tasks, occasionally report their progress to the JobTracker and ultimately return the result of the jobs’ execution to the JobTracker. Although MapReduce executes jobs in a reliable, fault-tolerant manner, the JobTracker, which represents the central authority of the system, runs on a single physical node and represents a single point of failure of the system. In case of JobTracker failure, all ongoing jobs are lost, the MapReduce system is brought into an undefined state and cannot continue normal execution. This can be a serious problem and it is unacceptable for long-running, time-critical jobs which must be completed within a specific time frame. In addition to the fault intolerance, running a single JobTracker in the system represents a constraint to the system’s scalability. Although there is a vast amount of work on the topic of solving the JobTracker fault tolerance and scalability issues, there is no solution reliable and efficient enough that could be adopted by the Apache Hadoop community.

The JBoss Infinispan is a transactional in-memory key/value NoSQL data store and data grid [5]. The data is saved in memory and could be asynchronously replicated to a configurable number of nodes, thus making the Infinispan a very fast system for storing highly available data. Operations on the data can be processed reliably using the Infinispan transaction system. Infinispan supports distribution of data amongst the nodes, which provides high scalability of the system in a situation where the amount of data rapidly grows. Infinispan also offers a distributed execution framework which can run a distributed task on the Infinispan cluster of nodes and provides the ability to detect task failure and initiate its failover.

In this paper we identify the requirements crucial for a JobTracker fault tolerance and scalability solution. Since it is a very fast, scalable and highly available data storage, we propose Infinispan to be used by the JobTracker to persist its internal state and reload it in case of failure, upon a
restart. We suggest running the JobTracker as a distributed task on the Infinispan Distributed Execution Framework, which will provide automatic failover of a failed JobTracker, out of the box. We suggest providing the scalability of the JobTracker by running multiple independent instances in the cluster. Load balancing component is implemented on the JobClient side, while the TaskTrackers can be dynamically (re)assigned to the running JobTrackers.

The rest of the document is organized as follows: in the Section II we give an overview of the research currently going on the topic of Hadoop fault tolerance and scalability. In the Section III we show the requirements identified for the solution and present its design. In the Section IV we describe the implementation details of the solution. Finally, in the Section V we present the tests performed on our solution and the results obtained.

II. RELATED WORK

There are many solutions and ongoing work on the topic of Hadoop fault tolerance and scalability that cover these issues to some extent. All of the solutions for the HDFS NameNode and MapReduce JobTracker fault tolerance suppose preserving the crucial state of the master server, which, in case of its failure, can be used by other system components to recover the system and continue with normal operation. Some of the solutions replicate the state to the slave components [6][7] while other save it to a persistent storage [8][9]. The approaches for the master restart vary from a manual restart [9] of the master to an automatic failover to another slave node [10][6]. In [8][9] the ongoing work is preserved after the JobTracker restart, while in [11] all the previously running jobs have to be restarted from scratch. The commercial solutions [10][6] offer stable fault tolerance functionality for the JobTracker and the NameNode, provided that they do not support the scalability of these components. On the other hand, most of the community solutions are still immature and as such not good enough to be incorporated into official Hadoop releases.

The main solutions for scalability rely on starting one master component per unit of work (e.g., job, application) [12][11] and thus limiting the responsibility of the master to only managing the lifecycle of its unit. This approach always requires adding an additional component which would become a new ultimate master of the system and whose role would be the management of the "sub-masters" and system resources. Besides the complexity of the implementation, this approach also transfers the problem of fault tolerance and scalability from sub-masters to the new master component.

Our solution represents a combination of the previously explained approaches that tries to cover as many of the basic requirements as possible for fault tolerance and scalability of the JobTracker.

III. ARCHITECTURE

The main goal of our solution is to provide an efficient implementation of the fault tolerant and scalable MapReduce JobTracker. Having this in mind, we identify the following crucial requirements imposed on our solution:

- **Fault tolerance** - The solution must continue functioning normally even when multiple simultaneous JobTracker failures occur, given that enough physical resources are provided.
- **Correctness** - In case the JobTracker restarts, the system must continue executing already running jobs correctly and produce correct final results.
- **Availability** - The system must be as available to the clients as possible in case the JobTracker fails.
- **Automatic failover and transparency** - The system should provide automatic crash detection and failover of the JobTracker to some other available node, without administrator intervention.
- **Scalability** - The system must allow JobTracker functionalities to scale, in case when the number of running jobs rapidly grows, given that enough physical resources are provided.
- **Performance** - The solution must not produce significant impact on the system performance in terms of the total jobs’ execution time and memory occupied by additional framework components.

**A. JobTracker Fault Tolerance**

The proposed system supposes running the JobTracker on a cluster of Infinispan nodes using the Infinispan DEF. Therefore, the system architecture contains four main components: Infinispan Cache instances, modified JobTracker, the Infinispan DEF master and the DEF failover policy.

Infinispan Cache instances form the Infinispan data cluster. Each Cache instance runs a new process on a physical node. Beside serving the data, it also represents a possible location for executing a DEF distributed task.

Infinispan DEF master is a component that submits a new task to be executed on the DEF, controls its execution and restarts it in case of failure. DEF master uses a failover policy in order to decide on which node of the cluster to restart the task.

The DEF failover policy is a helper component which is a part of the DEF master. The failover policy defines where to restart the JobTracker upon a failure, based on the current conditions in the Infinispan cluster, and each time a failover occurs the DEF master consults it. The failover policy also sets custom parameters that define the JobTracker specific behavior during a new startup.

The modified JobTracker represents a MapReduce JobTracker instance which is adapted to store its data in the Infinispan Cache and it is also suitable to run on the DEF as a distributed task. The main architecture components are shown in the Figure 1.

During normal operation, the Infinispan DEF master starts a new instance of the JobTracker on one of the Cache
nodes, already present in the cluster. Since the DEF master
dynamically decides the IP address on which to start the
JobTracker, the address is not publicly known, and the
JobTracker, after the startup, will have to publish it to the
cluster, storing it in the distributed Cache. This way it
makes itself discoverable by the other Hadoop components.
When choosing a node on which to start a JobTracker,
Infinispan DEF master sorts all available destination nodes
by the number of JobTrackers they already run and picks
the least loaded node. The cluster statistics is stored in the
Infinispan and it is updated by the DEF master each time a
JobTracker is started or failed over. Note that for providing
the fault tolerance of the JobTracker, the DEF master must
not start the JobTracker on the same physical node where
it resides, since if the node crashes both DEF master and
the JobTracker would be permanently lost. To overcome
this, when starting the whole system from scratch, we must
have a node where we will initialize start only an Infinspan
instance and it will serve as a destination node for running
a JobTracker in the future.

The JobTracker executes jobs received from JobClients
and delegates tasks to the TaskTrackers, like in a standard
Hadoop MapReduce framework implementation. Addition-
ally, the JobTracker stores all of its state in the local Cache
instance (located on the same physical node) which is
replicated to the other nodes later on, in order to provide
high availability of the data. Persisting the JobTracker data
structure in the Infinispan represents a first step towards
providing the JobTracker fault tolerance.

In a situation when the JobTracker fails, whether due to a
failure of the JobTracker’s process or the whole physical node,
the JobTracker DEF master will detect the failure, consult
the failover policy in order to determine the next JobTracker
node and restart the JobTracker on that node. In addition, the
JobTracker DEF master will migrate all of the JobTracker’s
state, from the other Cache node to the chosen node, in case
it is not already present there, thus making its data locally
available again. During the recovery startup, the JobTracker
will load its previous state from the local Cache instance
and continue with normal operation. The TaskTrackers and
JobClients will also detect the failure of the JobTracker using
their regular heartbeat mechanism, fetch the new JobTracker
address from the cluster and reconnect to it, whereupon they
can continue with their normal operation. This process is
shown in the Figure 2.

During the JobTracker downtime, all the MapReduce
jobs’ tasks already generated and assigned for execution to
particular TaskTrackers can continue with their execution
and their status will be updated when the JobTracker starts
again. Moreover, users can continue submitting new jobs,
since the new JobClients components created for these jobs
will continue retrying to connect to the JobTracker until it
starts again and then submit the user jobs to the JobTracker.

Using Infinispan DEF we totally delegate the effort for
detecting a failure and restarting the JobTracker to the
framework, thus relieving the work from a failure detector
implementation, which would be mandatory in case of a
Master - Slave JobTracker approach. In addition, keeping
only one JobTracker running in the system we eliminate
the need for leader election and any additional coordination
between the multiple JobTrackers. This design also supports
multiple simultaneous JobTracker failures, provided that
enough physical nodes are available, since there is one DEF
master component for each JobTracker in the system.

B. JobTracker Scalability

We decided to choose an easy and efficient way for pro-
viding the JobTracker scalability which considered running
multiple independent JobTrackers in the same cluster. In this
design, TaskTrackers are divided amongst the JobTrackers
and each of them receives tasks from a single JobTracker.
On the other side, JobClients submit new jobs and a load
balancing component distributes these jobs amongst the
JobTrackers currently running in the cluster. This is shown in
the Figure 3.

We assume that for an optimal operation of the system
there should be equal or more TaskTrackers than JobTrackers
in the cluster. Each JobTracker must have at least one Task-
Tracker assigned all the time and that will be a permanent
assignment. This is important in order to keep our load
balancing and TaskTracker distribution design working and
avoid gathering TaskTrackers around a single JobTracker.

1) TaskTracker Distribution: In many cases static dis-
tribution of the slaves amongst running masters cannot be
optimal, since the load of different master nodes varies over
time and we could get into a situation where some masters
and their slaves are idle while others are overloaded. In
order to overcome this problem we design an algorithm for dynamic assignment and re-assignment of TaskTrackers to JobTrackers.

A TaskTracker can be assigned to a JobTracker on two occasions: when it starts for the first time and when it needs to reconnect. When a TaskTracker initially starts, it first tries to find a JobTracker without any TaskTrackers already assigned. If there is a single one, it connects to it. If there are several JobTrackers, it connects to a random one. In both cases, the TaskTracker stores the JobTracker identifier as permanently assigned. If all the JobTrackers have at least one TaskTracker assigned, the TaskTracker finds a JobTracker with the highest load and connects to it. If there are more JobTrackers with the same load, it picks a random one.

A TaskTracker will change the JobTracker it is currently connected to only in the case when it is not a permanent assignment and the TaskTracker is idle for a specified time. A TaskTracker is idle if there are no running jobs in the MapReduce subsystem that contains the TaskTracker and its JobTracker, i.e. neither the TaskTracker reports any task in progress nor does JobTracker have a job which is not completed. When changing the JobTracker, the TaskTracker repeats the same procedure for finding a new JobTracker as when it was started for the first time.

2) JobClient Load Balancing: A simple load balancing algorithm was designed to distribute the users’ jobs amongst the JobTrackers running in the cluster. When a JobClient wants to submit a new job to the cluster, it first tries to find the least loaded JobTracker and submit the job to it. If the least loaded JobTracker has no TaskTrackers assigned (JobTracker started and not enough TaskTrackers), JobClient will skip it and take the second least loaded one. If there are multiple JobTrackers with the same load, the JobClient will pick a random one.

The JobTracker load is calculated as a sum of all of its running jobs’ loads. The load of a single job represents the size of its splits on the hard disk, that were generated by a JobClient. A possible alternative for calculating the load could be the number of map and reduce tasks that JobTracker jobs contain in conjunction with the number of available slots of the assigned TaskTrackers.

Since Infinispan is a replicated, highly available, persistent storage, we use it as coordination support when providing JobTracker scalability. JobTrackers currently running in the cluster hold their IP addresses in the Infinispan store so the TaskTrackers and JobClients can always discover their current locations. JobTrackers also regularly update the status of their current load and store it in the Infinispan. TaskTrackers and JobClients read it when implementing JobTracker assignment and load balancing procedure, respectively. The whole process explained in the previous two sections is presented in the Figure 4.

Using the simple load balancing and TaskTrackers re-assignment processes explained above, it is possible to achieve a better distribution of the running jobs and higher utilization of the cluster. That will speed up the jobs’ execution, lead to a smaller load of the running JobTrackers and provide for the scalability of the system. In the presented design, the JobTracker scalability can grow beyond any practical needs by simply adding a new physical node and starting a JobTracker instance on it.

IV. Implementation

The presented architectural design was implemented in the Hadoop MapReduce source code, 1.0.4. stable release, which is the most frequently used Hadoop release in big production environments such as the ones in Facebook, Yahoo and LinkedIn.

A. JobTracker fault tolerance

1) Saving the internal state: Due to a need for synchronization on some of the JobTracker objects in multiple places in the source code, we decided to leave all the JobTracker data structures in memory, which should be accessed in normal operation, and also on every update of the data, to persist them into Infinispan. In this way, we always have an up-to-date redundant copy of the JobTracker state objects and we can also control the access to these objects in the JobTracker code.
The JobTracker state we store in the Infinispan consists of the most of the JobTracker class fields. The fields that are not persisted are the objects that do not maintain the JobTracker management state required for the successful fail-over or the objects that maintain the state already persisted on HDFS.

The JobTracker class, as a main starting point of object serialization, is not serialized as such, but rather its substructures are. All the objects that need to be serialized implement the Java Serializable interface and in that way they become suitable for using the JBoss Marshalling framework. The main JobTracker data structures are objects that represent running jobs, their respective tasks and connected TaskTrackers. These objects are frequently accessed and they are contained in multiple collections in the JobTracker as well as in several other classes. In order to avoid loading a whole collection from the storage, when we only want to access a single object in it, as well as to maintain consistency of the object state, we keep the same object contained in different collections in a single place in the persistent storage. We store only the unique identifiers of the objects in a collection when we persist it, and the objects themselves we save as separate Cache entries, one for each object of the collection.

2) Transactions and locking: Every TaskTracker heartbeat can be seen as a cycle of JobTracker operations. After it receives it, the JobTracker updates its internal state based on the notifications about the progress of the tasks and generates the new tasks to be executed on that TaskTracker. All of these calculations are executed inside the JobTracker heartbeat() method which is called by a TaskTracker. In order to ensure the consistency of the JobTracker in-memory state with the Infinispan replicas and consequently the correctness of the JobTracker operations in case of a failover, we use the Infinispan transactions mechanism to provide a guarantee that all of the operations of the heartbeat() method are executed atomically, i.e. all of them or none. Thus, the whole heartbeat() method is put into a single transaction.

A significant flaw of the Infinispan transactions mechanism is that it does not support a transaction retry in case of a rollback. In our implementation that could be a big issue, since if a transaction aborts for some reason which does not include a JobTracker process crash, the state will be saved in the JobTracker data structures in memory but not in the Infinispan persistent storage. To avoid this situation, we use pessimistic locking with a big lock acquisition timeout in order to avoid deadlocks and unnecessary rollbacks.

In order to speed up the state saving process we use Infinispan asynchronous replication, which allows a caller method to return immediately after the data has been stored in the local Cache. The replication itself will happen in a separate thread in the background. The main drawback of the asynchronous replication is a potential inconsistency of the data on the other nodes. Nonetheless, since we access the data of the Infinispan cluster only from the JobTracker process, the inconsistency issue does not cause a problem in our case.

3) The JobTracker on the DEF: In order to be suitable to run on the DEF, the JobTracker has to implement DistributedCallable interface, which in its basics extends the Java Callable interface with the possibility to start a thread on the remote node. Two methods from the DistributedCallable interface that are required to implement are the call() method, which basically replaces the JobTracker main() method and setEnvironment() method, used to set the Infinispan Cache instance and custom parameters upon a failover. The custom parameters important for the JobTracker are: a boolean parameter indicating whether the JobTracker is restarted or freshly started, and the JobTracker port and its identifier, which are generated in the moment of the first start. To serialize JobTracker object we chose to implement a custom Externalizer, which provides us a control over the serialization process.

For the purpose of the JobTracker failover, two types of daemons have been additionally implemented:

- The IspnNode daemon starts an Infinispan Cache instance on some physical node and initiates or joins the Infinispan cluster depending on whether it is the first instance or not, respectively. This is the peer-to-peer running mode of Infinispan, where each Cache instance starts in the same JVM process as the IspnNode program. Besides the Cache instance, IspnNode also starts a Hadoop RPC Server listener on a specified port that waits for the TaskTrackers and JobClients remote requests for fetching the Infinispan data from the cluster.

- The JobTrackerDef daemon also starts an Infinispan Cache instance and a Hadoop RPC Server listener. However, the main feature of this daemon is that it starts a new JobTracker instance on one of the already running Infinispan nodes, using Infinispan DEF framework. After starting the JobTracker, JobTrackerDef holds a remote reference to the running remote JobTracker and, in case of failure, restarts it on some other, healthy node. For the purpose of a failover, the JobTrackerDef uses the JobTracker failover policy implementation of the Infinispan DistributedTaskFailover-Policy interface, which decides on which node to restart the JobTracker.

B. JobTracker Scalability

1) Running multiple JobTrackers in the Cluster: In our implementation, adding a new JobTracker to the system is as easy as calling the JobTrackerDef daemon, which starts a new JobTracker instance on one of the cluster nodes. Nevertheless, several adjustments have to be made in order to support multiple JobTrackers running in the cluster at the same time.

---

1We suppose that TaskTrackers and JobClients run on separate physical nodes, thus they cannot access the Infinispan cluster directly. For the remote access to the cluster, Infinispan provides the HotRod client - server architecture. Still, this kind of running mode cannot be mixed with the p2p mode.
Since JobTrackers have to be running on different nodes, their IP addresses are now assigned dynamically by the Infinispan DEF instance which starts and fails over the particular JobTracker. Since our system allows multiple JobTrackers to run on the same physical node, JobTracker ports have to be assigned dynamically as well, based on the number of JobTrackers already running on a particular Infinispan node. In our system the port is also assigned by Infinispan DEF instance.

The JobTracker http server, used to provide an HTML view of JobTracker operations also uses a fixed port and in the new implementation it is dynamically assigned as well.

The JobTracker uses a system directory, located on the HDFS, to store several system files. The path of the directory is configured statically in the "mapred.system.dir" configuration property, in the original distribution. In order to avoid overwriting the files in that directory by different JobTracker instances, the directory name is now customized for every JobTracker running in the cluster.

2) TaskTrackers Assignment and Load Balancing: A TaskTracker can change the JobTracker it is currently connected to only in case it is not a permanent assignment and the TaskTracker is idle for a specified time. If a TaskTracker does not get a permanent assignment, upon startup (and restart as well), it will start a separate thread, called IdleChecker thread, which serves to change the assigned JobTracker after a specified idle time. Specifically, the thread calculates the time interval in which the TaskTracker is idle. To check the status of the JobTracker jobs, the IdleChecker thread contacts the JobTracker every heartbeat time interval. After the maximum idle time expires, the IdleChecker thread will initiate a change of the JobTracker by checking the other JobTrackers from the cluster. If it finds a more loaded JobTracker, it will send the current JobTracker a special "leaving" signal in the next heartbeat, which will cause the JobTracker to issue the ReinitTracker action for the TaskTracker. After that, the TaskTracker will restart and re-initialize its internal state and repeat the same procedure for finding a JobTracker as it was started for the first time. The maximum idle time for a TaskTracker is defined as an additional parameter in the MapReduce configuration file.

The load balancing component is implemented in the JobClient code. Each time a user wants to submit a new job, the JobClient will contact the Infinispan to fetch the statistics of the currently running JobTrackers and find the least loaded JobTracker which has TaskTrackers already assigned. It will then submit the user job to that JobTracker. Every time a new job is received from a JobClient and its splits are generated, the JobTracker updates its load in the appropriate data structure in the Infinispan. When the job finishes, the JobTracker updates the load again.

C. Hadoop Configuration

Several configuration parameters that are important for the fault tolerance and scalability features are exported into an external MapReduce configuration file (mapred-site.xml), so they can be dynamically changed and can take effect without recompiling the source code. These are summarized in the Table I:

<table>
<thead>
<tr>
<th>Name</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mapred.infinispan.node.addresses</td>
<td></td>
<td>Comma separated list of IP addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>of Infinispan Cache instances</td>
</tr>
<tr>
<td>mapred.infinispan.node.port</td>
<td>6666</td>
<td>Infinispan node port for starting Hadoop RPC server</td>
</tr>
<tr>
<td>mapred.jobtracker.port.default</td>
<td>9990</td>
<td>Default port for JobTracker RPC server</td>
</tr>
<tr>
<td>mapred.tasktracker.idlemaxtime</td>
<td>30000</td>
<td>TaskTracker max idle time (in milliseconds)</td>
</tr>
</tbody>
</table>

Table I

Hadoop configuration

D. Changes to the Hadoop Execution Scripts

Since we developed two new Hadoop program daemons, IspnNode and JobTrackerDef, we had to update the main Hadoop execution script file, bin/hadoop, to add the ability for running them. Both can be started like all the other Hadoop daemons, i.e. "bin/hadoop ispnnode" and "bin/hadoop jobtrackerdef".

E. Infinispan Improvements

There are several differences in the startup process of a new and a restarted JobTracker. Thus every time the JobTracker starts or fails over, it needs to get some additional information from the JobTrackerDef instance or JobTracker failover policy, respectively. In the current implementation of the DEF this operation is not possible in the latter case (discussion at [13]).

This work makes a contribution to the Infinispan source code by adding a possibility to the failover policy mechanism to set a user specific parameters map which can be loaded from a user Callable later on. The changes have been incorporated into the DEF framework itself, believing that all the other DEF distributed tasks can benefit from it. The signature of the method added to the DistributedTaskFailoverPolicy interface is presented in the following snippet of code:

```java
Map<Object, Object> getEnvironmentParameters()
```

At the DistributedCallable side, the method setEnvironments was changed to read previously set parameters in the following way:

```java
setEnvironment(Cache<K, V> cache, Set<K> inputKeys, Map<Object, Object> params)
```

V. Evaluation

The main evaluation goal was to compare the solution with the official Hadoop release in terms of the performance and cost and to identify potential bottlenecks and overheads. All the experiments presented here have been performed in accordance to requirements presented in the Architecture section in order to show the level of their fulfillment.
A. Evaluation Methodology

Our evaluation addresses the following aspects of the solution:

- Robustness of the fault tolerant solution and the JobTracker restart time
- State saving overhead
- Memory consumption of the stored state
- Scalability of the solution

As experimental benchmarks we used a set of Hadoop programs from the HiBench Benchmark Suite [14], which consisted of Sort, TeraSort, WordCount and K-means test programs. For examining memory consumption we used YourKit Java Profiler [15] which provides creating memory snapshots.

All the experiments were performed on the cluster of machines with 8x 4-Core Intel 2.13GHz and Memory 40GB RAM each, running Ubuntu 64bit Linux.

Two deployment strategies have been used:

- For the original Hadoop distribution, the JobTracker and the NameNode were started on the same physical node, three DataNodes and TaskTrackers were started on three different nodes, each running a TaskTracker / DataNode pair.
- The fault tolerant and scalable implementation was running Infinispan nodes in addition. In this deployment, two physical nodes running Infinispan Cache instances were employed, forming an Infinispan cluster and one of them was also running a JobTracker.

B. JobTracker Fault Tolerance Efficiency

In this experiment we first wanted to show that after killing the JobTracker process the failover will really happen on another node and the JobTracker will continue executing all the running jobs correctly to the end; and second, to measure the time the JobTracker requires to perform complete restart and become operative again. In the experiment we ran 5 instances of the Sort job in parallel and killed the JobTracker process manually.

Each time we killed the JobTracker process it automatically restarted on another available Infinispan node and continued managing already running jobs to the end. In addition, the running jobs continued their execution on the TaskTrackers during the JobTracker downtime which we induced from the Job Clients status reports. We examined the outputs of the jobs and concluded that they were correct. We repeated the experiment 10 times and each time we got the expected behavior.

The total failover time of the JobTracker, as inferred from our implementation, can be divided into several categories, and the results by categories are shown in the Table II.

From the presented table we can see that the DEF master detects the JobTracker very fast and it needs approximately one second for that. This behavior was expected since the DEF master holds a remote reference to the running JobTracker. For loading the full state from Infinispan into internal data structures the JobTracker needs only one second, which is a result of the Infinispan in-memory data storing. For starting its internal services, JobTracker needs 7 seconds, and this time we can also consider as a fixed time, since it does not depend on any variable. The TaskTracker re-connection time shown includes the worst case maximum time that amounts 43 seconds and the expected maximum time which is 23 seconds. We considered the case when the TaskTracker heartbeat interval was 3 seconds, the TaskTracker was configured with the 10 retries when a connection problem with JobTracker occurred and the TaskTracker thread slept 1 second after each retry. Moreover, we assumed that the Infinispan node the TaskTracker was connected to also died with the JobTracker, thus, the TaskTracker had to reconnect to a new Infinispan node first. The retry period for connection to the Infinispan node was also 10 seconds with 1 second sleep between. The worst case scenario included additional 20 seconds in case there was a temporary network problem with the connection to the new Infinispan node and the JobTracker node. The presented maximum time for TaskTracker re-connection can be far smaller, since the number of retries and threads sleep time values can be reduced in the external configuration.

C. State saving Overhead

The main overhead in our solution for the JobTracker fault tolerance represents the state that is required to be saved in the Infinispan. We performed two types of experiments, first executing a single job of each of the several job types and the second executing a single job type and varying the number of its job instances executing in parallel. All the experiments were performed in a situation when no JobTracker failure occurred and we repeated each particular experiment 10 times and the numbers presented in the graphs are an average of all the executions’ results.

The settings for the HiBench jobs we used and size of the tasks’ inputs are shown in the Table III.

The results of the first experiment are shown in the Figure 5:

From the picture we can see that the original distribution shows better performances then the fault tolerant implementation, in terms of the execution time, and the difference represents the state saving overhead. The overhead was calculated as a percentage of the original time that needed to be added in order to provide fault tolerance. This is shown in the formula (1):
### Table III

<table>
<thead>
<tr>
<th>Job name</th>
<th>Input data size</th>
<th>Run settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wordcount</td>
<td>data size 3.2 GB</td>
<td>number of maps: 45, number of reduces: 70</td>
</tr>
<tr>
<td>Sort</td>
<td>2.4 GB</td>
<td>number of maps: 45, number of reduces: 70</td>
</tr>
<tr>
<td>Tera sort</td>
<td>100MB</td>
<td>number of maps: 180, number of reduces: 76</td>
</tr>
<tr>
<td>K-means</td>
<td>Num of clusters: 10, Num of samples: 20000000, Samples per input file: 4000000, Dimensions: 10, Max iterations: 5</td>
<td>number of maps: 300, number of reduces: 5</td>
</tr>
</tbody>
</table>

*Testing applications settings*

### Table IV

<table>
<thead>
<tr>
<th>Job name</th>
<th>Execution time (s)</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td>139.2</td>
<td>1.82</td>
</tr>
<tr>
<td>Wordcount</td>
<td>181.8</td>
<td>2.48</td>
</tr>
<tr>
<td>Tera sort</td>
<td>307.2</td>
<td>9.83</td>
</tr>
<tr>
<td>K-means</td>
<td>733.62</td>
<td>16.69</td>
</tr>
</tbody>
</table>

*Summary of the jobs’ executions time and overhead*

The overhead is the lowest in the Sort application and amounts to 1.82 percent. The job itself took about 2.5 minutes for both implementations. The highest overhead was found in the K-means application and amounted to 16.69 percent. The K-means job was the longest job and took about 10 minutes in total for the original solution and 12 minutes for the fault tolerant implementation. The overheads for all the jobs with the respective execution times are summarized in the Table IV.

From the summary and the experiment settings shown before we can conclude that the time overhead is directly proportional to the job execution time and the number of tasks processed during the job execution. This can be explained by the growing number of JobTracker task update operations sent to Infinispan and performed on every heartbeat received from the TaskTrackers. For small jobs, the overhead is negligible, as in the case of the Sort and Wordcount jobs, while for more complex jobs we can predict that the overhead can continue to grow with the increasing complexity of a job.

In the second experiment we chose one specific MapReduce application, Sort program, and varied the number of the copies of the application executing in parallel.

From the Figure 6 we can get a clear picture about the trend of the execution time when increasing the number of MapReduce jobs in the system. Although the total execution time increases for both distributions when the number of jobs arises, we can see that the difference between the two times slightly increases when the number of parallel jobs grows. As in the previous experiment, this can be explained by the number of the tasks update operations performed on the storage.

### D. Memory Consumption

The memory overhead which results from the MapReduce fault tolerance is defined by the memory size of the state the JobTracker stores in the Infinispan. We make multiple snapshots of the memory and calculate the total size of the objects the JobTracker produces and stores into Infinispan. The experiment is important in order to identified potential memory overloads that could cause system unavailability and even a loss of the currently executing jobs.

The deployment layout for this experiment supposed starting one additional Infinispan Cache instance on a separate physical node which served only for storing the JobTracker state. This was done in order to make it easier to extract objects of interest from all the Java objects located in the JVM heap. We used the Sort job and varied the number of parallel executing copies from 1 to 5. During the execution of the job(s) we performed snapshots every 30 seconds and...
analysed these snapshots offline. In addition, after a job finished, we continued making snapshots for 3 more minutes to see how the memory consumption changed.

From the Figure 7 we can infer that the size of the stored state for all the executions varies over the time and reaches its maximum at about half of the jobs execution lifecycle. The memory consumption is bounded between 2 to 16 MBs for all the executions. The leaps of the memory consumption are caused by the generation of new tasks during the jobs’ lifecycles, when their ID objects are added to the multiple data structures in the Infinispan. The memory falls shows that the JobTracker removes these objects from the storage after the tasks finish. We can observe the steep slope of the memory consumption at the end of each execution, where the state size falls to around 2 MBs. This is because the jobs’ executions are completed in that time and the job and task objects itself are removed from the storage. In the Figure 8 we show the maximum memory consumption reached during the jobs execution lifecycle for all the executions showed before.

In the Figure 8 we see that the memory consumption maximums go from 8 MBs for one executing job to 16 MBs when executing five jobs in parallel. From the presented logarithmic trend we can induce that the maximum consumption growth is smaller when increasing the number of running jobs in parallel, which makes us confident that we will not get a memory overflow when running additional parallel jobs.

E. JobTracker Scalability

In this test we wanted first to prove the correctness of our design implementation. Since we could not reach the bottleneck of the JobTracker scalability, due to testing environment hardware limitations, the second thing we decided to do in this test was to compare the performance of the scalable solution with the original distribution using a small set of running jobs.

We took a practical scenario where we have limited resources (in this case physical nodes) and measured two different cases: one where we utilized a resource to start a TaskTracker and the second where we utilized the resource to start the second JobTracker. We used four physical nodes for JobTracker(s) and TaskTrackers. In the first case, we started one JobTracker and three TaskTrackers and in the second case we started two JobTrackers and two TaskTrackers. In both cases each component was running on a dedicated node and we varied the number of running jobs from 1 to 5. For the experiment we used Sort job and measured the total execution time.

In our scalability solution, each re-run produced the same expected behavior; the submitted jobs were distributed amongst the two running JobTrackers and each of the two TaskTrackers was assigned to one of the running JobTrackers. The results obtained from the experiments are shown in the Figure 9.

On the diagram we can observe that the multiple JobTrackers deployment produces noticeably better performances than original distribution when executing two and four jobs in parallel and reduces the execution time for 24% and 16% respectively. On the other hand, for the execution of three parallel jobs we obtained worse results and an execution time similar to the single JobTracker fault tolerant deployment. The negative difference is caused by the state saving overhead of our solution.
The optimization obtained in scalable deployment can be explained by the fact that Hadoop MapReduce uses the JobQueueTaskScheduler by default, which schedules the submitted jobs one by one, waiting for previous jobs to complete in order to start the next one. This behavior is not acceptable for many jobs, when executing in parallel with other jobs. Using multiple JobTrackers we achieved a better level of parallelization by instantly scheduling the jobs which would otherwise wait for the previous jobs to complete.

VI. Conclusions

In this thesis we proposed, implemented and evaluated a design that solved the JobTracker fault tolerance and scalability problems in a way that did not significantly affect the overall system performances.

The proposed design is based on saving the JobTracker state into the Infinispan data store. The JobTracker is started on the Infinispan DEF framework as a distributed task, and in case the node where it runs fails, the framework performs the JobTracker automatic failover on some other, healthy node. Upon a restart, the JobTracker loads the stored state and continues with the normal operation. During the failover, TaskTrackers can continue executing the running jobs normally and the JobClients and TaskTrackers automatically re-connect to the JobTracker upon the restart. In addition, the system is available for users to submit new jobs all the time.

To provide the JobTracker scalability, the solution supports running multiple JobTrackers in the system in parallel. A simple load balancing algorithm is implemented on the JobClient side, which distributes users’ jobs amongst the JobTrackers currently running in the system. In the same time, TaskTrackers are distributed amongst the JobTrackers and each TaskTracker can be also re-assigned to another JobTracker in case the current JobTrackers load distribution requires that.

In order to fully support a custom failover of the JobTracker, minor improvements to the Infinispan DEF framework have been made and thus they can be applied to all other DEF distributed tasks.

The main drawback of the presented solution is the necessity of running one spare Infinispan node in order to provide proper behavior of the system, while the system is fully utilized after the first JobTracker failure happens. The second flaw of the solution is based on the shortcomings of the Infinispan DEF master, which is not fault tolerant.

The evaluation showed that the solution does not make significant a impact on the system performance. Memory consumption on the Infinispan nodes is controlled by the implementation which constantly removes finished parts of a job during its execution. The state saving overhead is negligible when running simple Hadoop jobs and grows slowly when increasing the number and complexity of jobs. The scalability part of the solution showed performance improvements under specific deployment settings when compared to the original distribution, which is mainly the result of the improved jobs scheduling.

REFERENCES


[2] Douglas Thain, Todd Tannenbaum and Miron Livny, Distributed Computing in Practice: The Condor Experience, Wiley Online Library, 2005


