Imperial College of Science, Technology and Medicine
Department of Computing

Dron: An Integration Job Scheduler

Ionel Corneliu Gog

Submitted in part fulfilment of the requirements for the
MEng Honours degree in Computing of Imperial College, June 2012

Contents

Chapter 1

Introduction

1.1 Motivation

The amount of digital information is growing at an astonishing pace. A study conducted by Gantz and Reinsel[1] correctly predicted that the amount of data created on the internet in 2011 will exceed 1.8 zettabytes. Thus, every month we created more data than the entire size of the Internet up to 2007.

The exponential increase of stored information has made data analysis a key research topic. The interest was also sustained by the need of many Internet companies to analyze the collected data sets in order to improve their products and sustain innovation. However, the increase in stored data has come to be a nuisance for some people. Traditional Relational Database Management Systems (RDBMS) can not scale up to meet the requirements. Moreover, even with all the advances in multi-core programming, the fastest machines could not handle up the processing of data.

The high costs of buying specially designed server together with the above mentioned reasons have encouraged Google, followed by many other companies, to develop an infrastructure based on commodity hardware. However, the new type of clusters also have drawbacks. For example, in a typical first year for a new Google cluster approximatively 5 racks misbehave causing up to 50% package loss, 20 racks fail making 40 to 80 machines dissapear for up to 6 hours. This has brought fault tolerance to the core of infrastructure development. As a result an abundance of scalable, failure tolerant computing frameworks have been developed (e.g. MapReduce[3], Hadoop[4], Dryad[5, 6]).

Given that the aforementioned systems were built for software engineers, they do not provide the abstractions needed by the layman users. Thus, more and more tools have been developed on top of the frameworks. Their purpose is to cover a new variety of use cases and to simplify usage.

Facebook has developed Hive[7] so that engineers can run SQL like queries on the 60+ Petabyte Hadoop cluster, Yahoo has developed Pig Latin[8] a combination of SQL queries and MapReduce style procedural programming. Moreover, Google has built many tools like Sawzall[9] and FlumeJava[10] around its MapReduce framework in order to conduct log analysis, store old data and perform graph algorithms as a series of chained MapReduce jobs.

With the abundance of frameworks and tools developed for specific tasks, engineers have started to create workflows that make use of multiple systems. As presented in [11], Facebook uses an entire suite of systems in order to conduct data analysis. For example, Scribe[12] is used to move data from their web tiers to HDFS[13]. Moreover, jobs that scrape the federated MySQL databases, containing all website related data, are run on daily basis. Lastly, data is transformed into Hive readable format and log analysis is conducted. However, since the amount of data copied varies from hour to hour, data analysts do not have an easy way to manage their workflows and to conduct their experiments.

In this report we will describe a new system called Dron that will attempt to fill in the above described gap. The purpose of Dron is to make frameworks more interoperable by providing an integration job scheduler that can handle job dependencies. The scheduler will allow users to define large workflows that make use of multiple systems. Moreover, it will display information about data and jobs so that similar steps within different workflows will be identified and performed only once.

1.2 Contributions

This project made the following contributions:

- **Developed New Job Scheduler**
  We developed a highly scalable and fault tolerant job scheduler that is able to handle hundreds of thousands of jobs a second, distributed over few thousands machines.

- **Integration of Multiple Frameworks**
  We demostrated the easiness of adapting current infrastructure systems to cooperate with Dron for job scheduling.

- **Removed Duplicated Computation Within a Data Center**
  By placing Dron at the core of every job computation, we allowed it to capture the entire graph of job and resource dependencies. Thus, users can avoid conducting the same computation more than once.

- **Opened Opportunities for New Systems**
  By bringing the state of the art frameworks together, in a data center, and by providing a way of creating complex workflows, we allow for new computations and systems (e.g. a recommendation engine) to be expressed as a series of Dron jobs.

- **Fault Tolerance Testing**
  We proved the well functioning of the system under various failure scenarios and discussed its shortcomings.

- **Scalability Evaluation**
  We evaluated the scalability of Dron by performing a variety of tests whose aim was to measure the behaviour of the system under different types of load. Moreover, we compared it with other state of the art workflow managers.

1.3 Outline of this report

We continue with Chapter 2 where we first familiarize ourselves readers with the current existing solutions for running large process or data intensive computations. While preseting each framework we also briefly discuss its use cases. The second part of the chapter is devoted to other state of the art workflow job schedulers.

Following, in Chapter 3 we give an overview of the highly scalable and fault tolerant system we developed to tackle the existing problems. We do so by providing few use cases, explaining and demonstrating its usage.

In Chapter 4, we focus on the several implementation iterations we conducted. Moreover, we will be explaining the rationale behind our current system design. Specific issues encountered while developing the system are also discussed.

Chapter 5 demonstrates the easiness of adapting frameworks to work with the new job scheduler. First, we go over the possibilities of integration provided by Dron. Subsequently, we exemplify the changes we had to make to each system.

In Chapter 6 we discusses the fault tolerance of our scheduler by analysis its behaviour under several failure situations. Moreover, we also describe the shortcomings Dron has and provide reasons for their existence.

Subsequently, Chapter 7 exemplifies the opportunity of building new systems on top of the job scheduler. We show the development of a recommendation engine using Dron and several of the frameworks we adapted.

In Chapter 8 we evaluate the performance of Dron by conducting several tests that stress its ability to handle a large volume of different jobs. Moreover, we also use data provided by Facebook to analyse the system's suitability for current industry needs.

Lastly, in Conclusions (Chapter 9) we provide a retrospective of the project, analyse the goals, mention the lessons we learned and outline the areas for future work.

Chapter 2

Background

Workflow scheduling and management has been a perpetual research topic since the invention of the first batch computer, up to grid computing and distributed software frameworks. Even tough, much research has been conducted in the area we believe that there still are plenty of opportunities once with the development of the new large cluster computation frameworks.

In the first few sections of the chapter we will briefly explain how several current frameworks work. They will help us understand how large data analysis is conducted and why there is a genuinely need for a new framework management system. Moreover, the first few sections will give us the background required to analyze current state of the art alternatives to Dron.

2.1 MapReduce

The concepts behind MapReduce were introduced in a paper by Dean and Ghemawat[3]. It is a *programming model* associated with a highly scalable framework for computation on large data sets. As the authors acknowledged, the abstractions behind MapReduce were inspired by the map and reduce primitives present in Lisp and many other functional programming languages.

A MapReduce job receives as input a set of key/value pairs and returns as ouput a possibly different set of pairs. The framework does not restrict the types of the input and output. Thus, users can pass to a job from simple pairs of integers up to complex files location.

MapReduce conducts the processing with the help of two functions provided by the users: *Map* and *Reduce*. The *Map* function receives as input a key/value pair and outputs a list of possibly different key/value pairs. The function is used in the first step of the computation by being applied to every single key/value input pair. Following, the job goes into an intermediate state in which the framework groups the values emitted by the *Map* function (i.e. all the values corresponding to a key are merged into a pair of key, list of values). Lastly, the *Reduce* function, also written by the user, is applied on every pair resulted from the intermediate step. It receives as input a key/list of values pair and its purpose is to merge the values to form a possibly smaller list. The input of the *Reduce* function is supplied using an iterator. Thus, allowing the framework to handle lists that do not fit into the machine's

memory.

The programming model behind MapReduce (i.e. splitting the computation into a series of smaller steps) is perfectly suited to a world in which machine failures are common. Whenever a machine crashes, or even worse, a switch misbehaves, the framework can simply restart the affected map/reduce tasks on different hardware. Moreover, the system can even preemptively duplicate several tasks so that the duration of job does not get affected in case of failures. The aforementioned features have made the framework to be suitable for deployment on a cluster of commodity hardware.

Together with MapReduce, Google has developed **Google File System (GFS)** [27]. It is a highly scalable, fault tolerant distributed file system suitable for storing large amounts of data. Since one of its requirements was to perform well on large sequential reads, the architects have decided to only support large append-only files. Moreover, these are divided into data chunks of 64Mb or more. The chunks are replicated on several (usually 3) commodity machines. This gives the file system the power of supporting machine or even more complex network failures. Having the aforementioned features has made GFS the traditional storage solution for the inputs and outputs of the MapReduce jobs run within the company.

We will not go into further details about the two pieces of infrastructe described above. However, we would like to emphasize the impact they had over the industry. They have inspired engineers to build an entire suite of similar open source systems. In the following section we will cover some of the publicly available solutions.

2.2 Hadoop Ecosystem

Inspired by the paper published by Dean and Ghemawat[3] Yahoo has started a similar open source project called Hadoop[4]. Subsequently, many other major Internet companies (e.g. Facebook, Linkedin) have joined the project and embraced it as a crucial piece of infrastructure. However, not long after the development, the engineers have quickly started to feel the need of new abstractions on top of the low-level MapReduce programming paradigm. As a result, they have created an entire suite of libraries and systems that run on the Apache Hadoop framework. The following subsections will describe several of the products that have been built and their usecases.

2.2.1 MapReduce

The open source community together with several software companies have managed to create an equivalent to Google's MapReduce framework. As described in Section 2.1 the computation is conducted with the help of two functions provided by the users: *Map* and *Reduce*.

In order to get a better understanding of the framework's usage, we will go over a simple example. We want to count the number of occurences of every word within a large file. In order to do so, we take each line from the file and build a pair out of its number and content. Finally, we pass it in as input to the *Map* function provided in Listing 2.1. The function splits the line into words and subsequently emits for each word a pair having the word as key

and one as the value.

```
1  public void map(LongWritable key, Text value,
2      OutputCollector<Text, IntWritable> output, Reporter reporter)
3      throws IOException {
4    IntWritable one = new IntWritable(1);
5    Text word = new Text();
6    String line = value.toString().toLowerCase();
7    // Split the line into words.
8    StringTokenizer tokenizer = new StringTokenizer(line);
9    while (tokenizer.hasMoreTokens()) {
10     word.set(tokenizer.nextToken());
11     // Generate a new intermediate pair consisting of word and 1.
12     output.collect(word, one);
13     reporter.incrCounter(Counters.INPUT_WORDS, 1);
14   }
15 }
```

Listing 2.1: Word Count Map Function

In the intermediate step the framework sorts the pairs emitted by all the *Map* functions. Subsequently, it groups together all the pairs that have the same key (i.e. emitted for the same word) into a key/list of values pair. Every pair resulted from the intermediate step is given as input to the *Reduce* function provided in Listing 2.2. The function simply counts the number of values from the input pair and emits it as the output.

```
1  public void reduce(Text key, Iterator<IntWritable> values,
2      OutputCollector<Text, IntWritable> output, Reporter reporter)
3      throws IOException {
4    int sum = 0;
5    while (values.hasNext()) {
6      // Add up the number of intermediate values existing for each word.
7      sum += values.next().get();
8    }
9    // Output the number of occurences of each word.
10   output.collect(key, new IntWritable(sum));
11 }
```

Listing 2.2: Word Count Reduce Function

An overview of the framework's architecture is presented in Figure 2.1. The system consists of a single master node called *JobTracker* and a set of slave nodes on which *TaskTrackers* are run. The master is responsible for scheduling tasks (i.e. map and reduce functions) onto the slave nodes. Moreover, while assigning tasks, it tries to locate the nodes that contain the data required by the computation. Furthermore, it also uses a heartbeat protocol to keep record of the free slots every *TaskTracker* has.

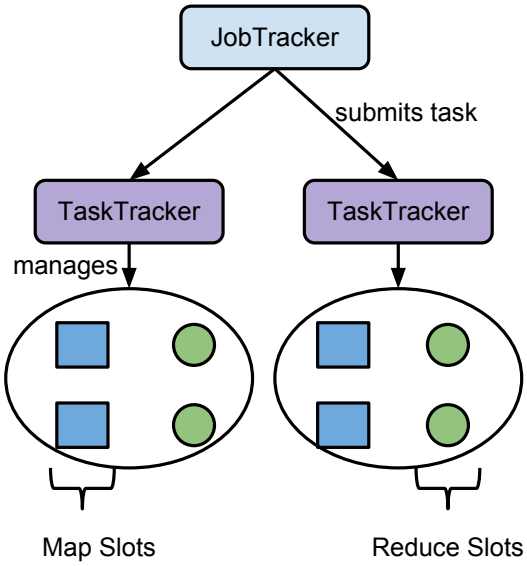

Figure 2.1: Hadoop MapReduce Overview

2.2.2 Hadoop Distributed File System

Similar to Google's solution, the open source community has built Hadoop Distributed File System (HDFS)[13]. It is a file system built to run on commodity hardware and to serve as a storage layer for the plethoria of Hadoop applications. The system was designed to meet the following goals:

- **Fault Tolerance** - hardware failures are common in large clusters. As Dean presented in [2], in the typical first year of a new cluster there are approximatively 20 rack and thousands of hard drive failure.
- **Streaming Data Access** - the purpose of the system is to support large data analyzing MapReduce jobs. Thus, the emphasis was set on high throughtput rather than low data access latency.
- **Optimize Data Locality** - it is more costly to send the data to the computation node than running the computation where the data is.

We believe that the file system has managed to partially meet the abovementioned requirements with the help of the master/slave architecture pictured in Figure 2.2.

The NameNode is the key component of the system. It manages the file namespace and administers the access to the files. The other important piece of the system is the DataNode. Every node that is part of a cluster is running one. Its purpose is to manage the data stored on the node.

HDFS's requirements have encouranged the developers to model the large files as a series of 128 or 256Mb blocks. The blocks are managed and replicated by the DataNode across

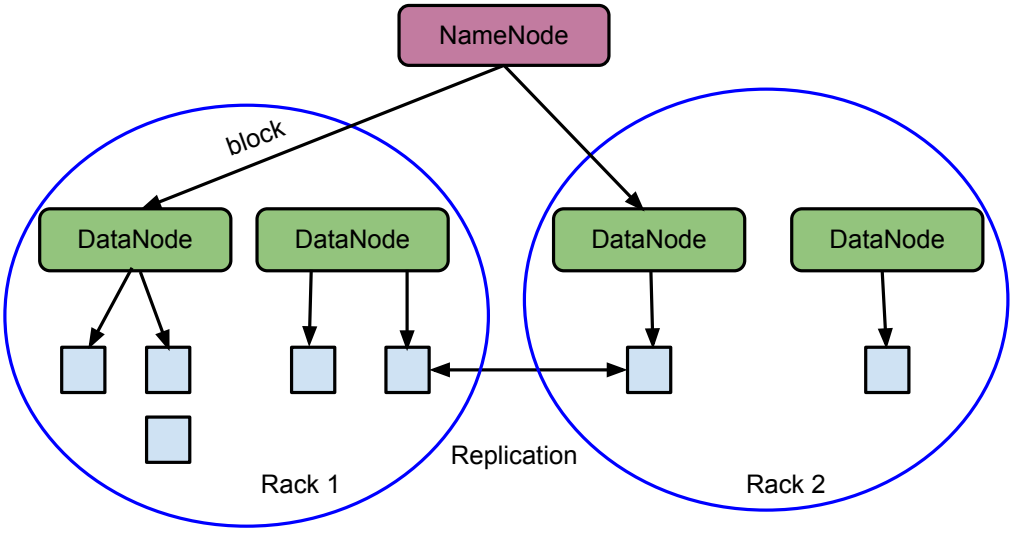

Figure 2.2: HDFS Overview

machines and racks, at the instruction of the NameNode. Replication has allowed the file system to provide a high degree of fault tolerance. However, as we can see in Figure 2.2, the NameNode is a single point of failure. Thus, the crash/restart of the machine that is running the NameNode will suddenly make the whole file system unavailable. Moreover, the NameNode has proved to be a bottleneck in the large clusters managed at Facebook. As a result one of the top efforts in the community is to improve its performance and to make it highly available.

2.2.3 Mahout

Mahout[15] is a machine learning library built on top of the Hadoop MapReduce framework. It provides implementations for clustering (e.g. group related news articles), classification (e.g. tag documents based on their content) and batch based collaborative filtering (e.g spam detection). The algorithms are implemented as an iterative sequence of MapReduce jobs that are executed on the Hadoop framework. As we will see in Chapter 7, Mahout and Dron will help us to easily build a recommendation engine.

The success of the Mahout library has also started to expose some of the weakpoints of Hadoop's Mapreduce framework. A commonly discussed problem is its poor performance in running iterative jobs (i.e. the majority of Mahout's jobs). This is due to the fact that Hadoop is writing the output of each MapReduce job to the distributed file system and subsequently reads it back on during the the next iteration. As we will see in Section 2.4, the research community has built many frameworks that address the issue.

2.2.4 Dron and Hadoop

The importance of the systems we have presented above has convinced us to put a considerable amount of work into adapting them so that MapReduce and Mahout jobs that depend on HDFS files can be run with the help of Dron. In Chapter 5 we give details the changes we have made.

2.3 Query Engines on Top of MapReduce

2.3.1 Hive

Hive[7] is an open source data warehousing system running on top of Hadoop. It was developed by Facebook to provide an easier to use interface for their data analysis team. This was accomplished by supporting a SQL-like declarative language which transforms queries into MapReduce jobs. As a result of its success, Hive has continuosly challenged the scalability of the Hadoop framework. For example, Facebook currently manages a HDFS/Hive cluster containing more than 70 petabytes of data.

The SQL-like language provided by Hive is called HiveQL. It currently supports data definition statements (DDL) to create or delete tables. When creating a new table the users have to specify the types of the columns. Hive currently supports columns of primitive types, arrays, maps and compositions of the aforementioned. In order to get a better understanding, we create in Listing 2.3 a table of page_views that stores the page accesses made by the users of a website. It consists of three columns, the first one keeps the viewing time, the second one stores the user id and the last one contains the page URL. Moreover, as the table can quickly grow to a notable size, we decided to partition it over the days we are logging the information.

```
1 CREATE TABLE page_views(viewTime INT, userId BIGINT, pageUrl STRING)
2 PARTITIONED BY(ds STRING)
3 STORED AS SEQUENCEFILE;
```

Listing 2.3: Example of a Hive Data Definition Statement

Additionally, HiveQL provides load primitives so that users can directly import data from their HDFS deployment. The language also includes different options of accessing and aggregating the data. For example, in Listing 2.4 we show an example that filters out the vists made by the user with id 42.

```
1 SELECT page_views.*
2 FROM page_views
3 WHERE userId = 42;
```

Listing 2.4: Example of a Hive Query

The data managed by Hive is stored in the Hadoop Distributed File System (HDFS). Each table has a corresponding directory. Moreover, each table can have one or more partitions that are modelled as subdirectories. The files within these directories store the data that has

been serialized by the default builtin serialization library or by implementations that have been provided by users.

Another notable feature of Hive is the system catalog called Metastore. It provides metadata about the framework and the data it manages. For example, for each table, it stores its location, list of columns and their types, serialization/deserialization information and much more. The metadata is used by the HiveQL to create the MapReduce jobs. Moreover, it can also be accessed by external services via Thrift[36] (a cross-language framework for services development) to determine the existence of certain tables or partitions.

By any means, Hive does not provide all the features of a traditional Relational Database Management System (RBDMS). For example, as a consequence of being built on top of HDFS, the query language does not support updating and deleting rows in existing tables. The reason behind this shortcoming is that the file system was built to be able to scale up to petabytes of data. Thus, it only supports append-only operations making it unsuitable for storing small quantitis of data.

2.3.2 Pig

Pig[8] represents Yahoo's attempt at closing the gap between the declarative style of SQL and the low-level programming model of MapReduce. Similar to Hive, it provides a language (PigLatin) which gets translated into a series of MapReduce jobs. A PigLatin program expresses the computations as a series of steps with each one carrying out a single data transformation (e.g. filtering, grouping, aggregation). As Hive and Pig are two systems that are very similar we decided that there is no reason for integrating both of them with Dron. We choose to adapt Hive, hence we will not dwelve into more details about Pig.

2.3.3 Google's Solutions

Data analysis has been at the core of Google sint its early existance. Thus, over the past seven years they have published papers about three highly scalable query execution engines. Following we will briefly describe each one of them.

Sawzall

Sawzall[9] is a procedural domain-specific programming language developed on top of Google's MapReduce framework. The design of the system was influenced by two observations. Firsly, the order in which the records of a table are processed is unimportant if the querying operations are commutative across them, and secondly, the order in which the records are processed is unimportant if the aggregation operations are commutative.

This has encourage the developers of the system to break the computation into two phases. In the first step the framework runs a Sawzall script over each record of the input. The script can output only by emitting data that is sent to an external aggregator. The role of the aggregator is to gathers the results from each record and processes them. The authors of the system have decided to conduct aggregation outside of the language in order not to expose parallelsim to the Sawzall users and to safe-guard the infrastructure of poorly

implemented aggregators. As a result, Sawzall only offers a collection of aggregators (e.g. maximum, quantile, sum) that have been carefully implemented.

Dremel

Not long after the launch of Sawzall, users have started to feel the pain of waiting for a long time for their scripts to run. Thus, a new scalable, interactive ad-hoc query system for analysis of read-only data, has been built. Dremel[35] achieves the abovementioned features by combining multi-level execution trees and columnar data layout. It is capable of accessing data stored in Google File System or in other storage layers such as Bigtable [37]. Moreover, Dremel runs queries natively (i.e. it does not translate them into MapReduce jobs). As a result, it is able to execute queries over a trillion records in just several seconds. Sadly, we were not able to integrate Dremel with Dron as there is no open source implementation available at the moment.

Tenzing

Tenzing[34] is yet another effort to build a query engine for ad-hoc data analysis on top of MapReduce. It is an almost complete SQL92 implementation , highly efficient with a latency of as low as ten seconds. In order to achieve this performance, Tenzing optimizes both the queries provided by engineers and the resulted MapReduce jobs. By developing it tightly coupled with MapReduce, the engineers were able to avoid spawning new binaries for each Tenzing query, implement streaming between MapReduce jobs, disable the MapReduce sorting stage for some queries and many more. Similar to Dremel, we were not able to experiment with the system as currently there is no open source implementation.

2.4 Iterative Computation

While introducing the Mahout library in Subsection 2.2.3 we mentioned Hadoop's poor performance in running iterative MapReduce jobs. As a result, many have set on creating new frameworks (e.g. Spark[16, 17], HaLoop[18], Twister[19] that are more suitable for the given usecase. In the following subsections we will go over a fraction of the solutions that have been developed.

2.4.1 Spark

Spark is the result of the observation its authors have made: there is an entire set of applications that can not be efficiently expressed under the acyclic programming model provided by standard MapReduce (e.g. iterative jobs, interactive analysis). It is a framework implemented in Scala, that fixes the aforementioned shortcomings with the help of two new abstractions: resilient distributed dataset (RDD) and parallel operations on these datasets.

The RDD is a read-only collection of objects partitioned across a set of machines. Users can construct RDDs from the files of a distributed file system (e.g. HDFS), from splitting a Scala collection into slices that will be distributed over the nodes or by applying a flatMap

operation over an existing RDD. The computation can be performed on RDDs with the help of the following parallel operations:

- **reduce** - combines dataset elements using a provided function.
- **collect** - sends all the elements of the dataset to the user program.
- **foreach** - applies a provided function onto every element of a dataset.

Furthermore, Spark provides two types of shared variables: *accumulators* and *broadcast* variables. They are the result of the fact that operations like map and reduce involve passing closures to Spark. Thus, in order to avoid sending variables to every worker node, programmers can use *broadcast* variables which are distributed to the workers only once. On the other hand, *accumulators* are variables onto which workers can only apply associative operations.

The features provided by the framework have encouraged us to study it more and to make several changes so that it can be used with the Dron scheduler. In Chapter 5 we will explain in detail the changes we have made.

2.4.2 HaLoop

HaLoop is a modified version of the Hadoop MapReduce framework that on average reduces iterative's jobs runtime by 45%. The improvements are the result of two observations that have been made by the authors of the system:

- The termination condition of the iteration, may involve detecting when an approximative fixpoint has been reached. Thus, the check of the condition may require an additional MapReduce job for each iteration.
- Even tough the data between iterations is almost unchanged on many instances, Hadoop still writes outputs into HDFS and reads them back again during the next iteration.

HaLoop solves the above mentioned problems by making the following key improvements:

- Stores the output of the data on the disk of the machine where the computation was executed. On the next iteration, it tries to allocate the map/reduce functions on the same machines as they were in the previous step. In doing so, HaLoop avoids writting to HDFS and implicitly waiting after the entire slow replication protocol.
- Requires the user to implement several functions that will be used to determinate the convergence to an approximative fixpoint. For example, the user has to provide a function that computes the distance between two return value sets sharing the same out key.

We believe that even tough HaLoop and Spark are based on different concepts, they can be used to solve the same set of problems. Thus, having already integrated Spark with Dron we decided not to devote any extra time to HaLoop.

2.5 Non-MapReduce Systems

As MapReduce has started to be adopted more and more, many have started to feel that its programming model is too restrictive. Hence, they have set onto building frameworks that can express more general computations. We will briefly describe several systems out of the plethoria that have been built (e.g. Dryad [5, 6], Pregel[22], Percolator[42], BOOM Analytics[38], CIEL[21]).

2.5.1 Dryad

Dryad is a general-purpose distributed execution engine designed for data intensive applications. A Dryad job is modelled as a dataflow graph. Vertices represent the computation points, while the directed edges model the communication channels (e.g. files, TCP pipes, shared-memory queues).

The users have to provide implementations for each type of vertex. They usually are simple sequential programs. Despite that, concurrency is supported on Dryad by being able to run non-dependent vertices at the same time. The authors of the system have also implemented a language that makes it easy to specify common communication patterns. As we can see in the graphs presented in Figure 2.3 there are four basic operators:

- $A \hat{} n$ - creates n copies of the vertex A.
- $A >= B$ - creates a directed connection between A and B. Every outgoing edge from A is assigned in a round-robin manner to B's inputs.
- $A >> B$ - forms the complete bipartite graph between A's outputs and B's inputs.
- $A \mid\mid B$ - merges two graphs. It is useful for constructing communication patterns such as fork or join.

We believe that some interesting researh has been conducted in conjuction with Dryad. For example, DryadInc[43] is a system that tries to almost automatically reduce the amount of duplicate analysis. It extends Dryad's Job Manager with a logic that detects duplicate computations and rewrites the directed acyclic graph accordingly. Optimizing computation within a cluster is one of Dron's goals as well. However, we will not be able to provide an automatic solution to the problem as Dron is working in a much more heterogeneous environment (i.e. a whole suite of frameworks). Nonetheless, we forsee the posibilities of extending Dron to support such a feature.

Lastly, we would like to point that Dryad was intended to act a foundation for other frameworks that would be developed on top of it. However, unknown reasons have determined Microsoft to stop the project and to move on to developing their own Hadoop-like system.

2.5.2 CIEL

CIEL [21] is a universal execution engine for distributed data-flow programs. It supports a dynamic task graph that can adapt based on the data it processes. The system has been

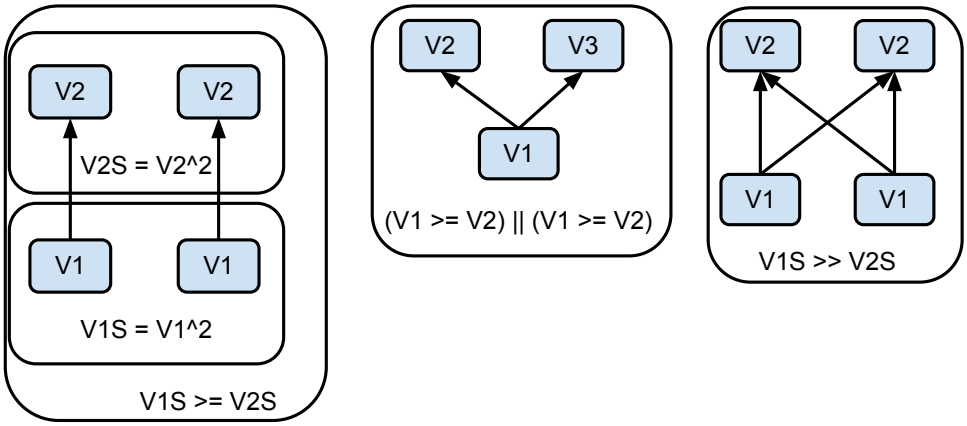

Figure 2.3: Dryad Operators

developed in conjunction with Skywriting [20], a Turing complete scripting language suitable for expressing iterative algorithms.

CIEL uses *objects*, *references* and *tasks* in order to express the computation. *Objects* are the output of a job, they are uniquely named, unstructured, finite-length sequences of bytes. Similar to the majority of object oriented programming language, *objects* can be used without owning their full content. This is achieved with the help of *references* (i.e. pairs of name and set of object locations). With the help of Skywriting the computation can be expressed as a series of *tasks* that execute on a single machine. These *tasks* can also have dependencies on *references* (i.e. they will only run when the data pointed to by the *reference* will be available).

CIEL tasks can perform two actions that affect the result of a computation:

- **Output** - they can publish one or more objects.
- **Adapt** - by spawning new tasks that perform additional computation. In this way, an execution graph dependant on the processed data can be constructed.

Finally, we would like to point that we appreciate the additional expressivity given by Skywriting and the CIEL framework. However, we believe that as jobs will get more and more complex it will be difficult to debug them and to track the execution of the tasks.

2.6 Mesos

As we have already seen, many frameworks have been developed. Nonetheless, we believe that many more will be created as there will not be an optimal system to solve all the problems. This also was the reason for which a team of researchers has built Mesos[23, 24], a platform for sharing commodity clusters between different computing frameworks. Its purpose is to reduce the duplication of data, improve utilization of resources by collocating systems within the same cluster and reduce the costs of sending data between clusters.

Mesos is a two-layer scheduler that does not take the control over scheduling from the other systems. It manages to accomplish this by introducing the concept of a resource offer (i.e. a set of resources that a framework can use on a maching to run its tasks). On every scheduling loop Mesos tries to keep fair sharing among frameworks by providing them resource offers. Each framework can decide to accept or refuse the offer if it does not suit its needs (e.g. data locality). By refusing a resource, the framework still remains under its fair share, thus the cluster manager will offer resources to the same framework during the next scheduling iteration.

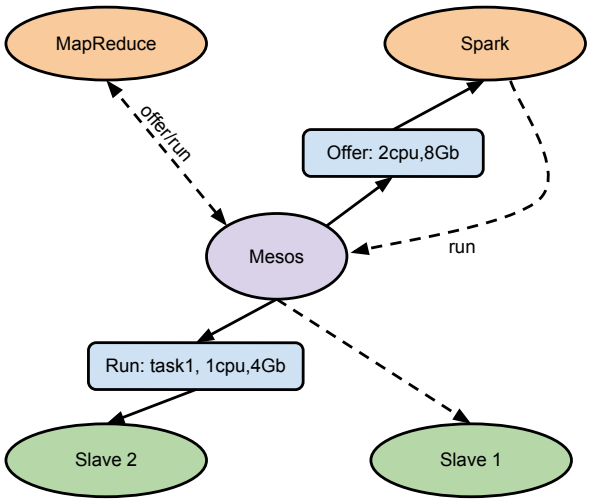

Figure 2.4: Resource Offering Overview

We believe that once with the exponential growth in stored data, companies will find more and more difficult to manage separate clusters for each type of framework. This will make Mesos or a similar system a crucial piece of infrastructure. By acknowledging this, we decided to use Mesos together with Dron for better resource sharing. Thus, the schedulers of the majority of the frameworks we adapted collaborate with Mesos for better resource utilization. To sum up, we think that a combination of Mesos and Dron may represent a base for conducting large scale data analysis.

2.7 Related Work

We believe that having analyzed several of the current state of the art systems , we have the background knowledge to proceed to the following sections. Moreover, we have also got a better understanding of why we believe an integration scheduler is required. The remaining of the chapter will be devoted to current workflow (data pipelines) management systems.

2.7.1 FlumeJava

FlumeJava[10] is a Java library developed by Google with the purpose of reducing the complexity of development, testing and running of data parallel pipelines. The core of the library contains several parallel collections together with few parallel operations.

The parallel collections do not expose details about how data is stored. Thus, users can develop and test their pipelines by storing a small data set into the memory. As a result, they will not be required to adapt the data pipeline whenever they will want to run it on the entire data set. Similar to collects, parallel operations abstract their execution strategy (i.e. an operation can run locally or as a MapReduce job). Tables 2.1 and 2.2 give a short description of the collections and primitives operations available in FlumeJava.

Collection	**Description**
PCollection ⟨T⟩	an immutable bag of elements of type T. The elements can be ordered (sequence) or not (collection)
PTable ⟨K,V⟩	a immutable multi-map with keys of type K and values of type V

Table 2.1: FlumeJava's Collections

Primitive Operations	**Description**
parallelDo()	map a function over each element of a PCollection and returns another PCollection
groupByKey()	converts a PTable into a collection of type PTable ⟨K, Collection⟨V⟩⟩
combineValues()	takes as input a PTable⟨K, Collection⟨V⟩⟩ and an associative combing function on Vs, and returns a PTable⟨K,V⟩
flatten()	flattens a list of PCollections into a single one containing all the elements of the input Collections.

Table 2.2: FlumeJava's Operators

The library's parallel operations are not actually run when they are invoked. Instead they are added to an internal execution plan graph. Once the entire graph has been constructed, FlumeJava proceeds onto optimizing it. Moreover, when the operations are actually run, the library chooses how to execute them (i.e. a loop run locally vs. a MapReduce job) based on the size of the input and the latency a MapReduce job involves.

Finally, we acknowledge that FlumeJava has managed to accomplish its goals. However, we think that providing the functionality as a library imposes several limits. For example, the designers have restricted the users to a specific suite of languages (i.e. JVM based). Moreover, the users are required to find a 'perfect' machine onto which a FlumeJava process can remain active for as long as the computation lasts. Otherwise, if the machine on which the application is running, crashes or restarts, the whole dataflow is lost.

2.7.2 Oozie

Oozie[30] is a workflow scheduler developed by Yahoo for the Apache Hadoop framework. An Oozie workflow is a directed acyclic graph composed of two different types of nodes: *action* and *control*. The former type starts jobs in external systems (e.g. MapReduce, Pig jobs) or it can simply run a Java program on a remote machine. The later type of node is used to control the execution path of the workflow (e.g. fork, join, decision nodes). For example, the *fork node* is used to split the execution path (i.e. run two or more different jobs) and the *join node* is used as a barrier (i.e. the next job will not run until all the above running jobs have completed).

A workflow is defined in an XML file that is stored in an HDFS directory together with all the files required for each node (e.g. Pig scripts, MapReduce job jars). Subsequently, users run the workflow by using a provided command line tool. Finally, they can monitor the execution of their job via a Web UI.

To get a better understanding of Oozie's capabilities, we will go over a simple example. We want to build a workflow that scans a document and calculates the number of occurences of each word. Moreover, we also want to count the number of distinct words. We can express the computation as a two node Oozie workflow. Listing 2.6 describes a word count MapReduce job. The creator of the action node must minimally specify the map/reduce classes to be used, the input and the output directory, and finally the actions to be conducted in case of failure or success.

Subsequently, the creator of the workflow can specify a Pig job that counts the number of unique words. Similar to the MapReduce action node, the Pig node must define the location of the script to be run, the input and output parameters and the nodes to be executed in case of failure or success.

Since machine crashes/restarts are common in clusters of commodity hardaware, Oozie also provides a basic mechanism for handling workflow failures. Users are allowed to rerun their jobs. While doing so, they have to specify a rerun policy. They can either inform the scheduler which nodes it must skip or allow it to automatically rerun the workflow from the failed node. However, users must also ensure that all the workflow cleaning operations have been conducted before reruning the computation.

```
<action name="WordCount">
  <map-reduce>
    <job-tracker>\$(jobTracker)</job-tracker>
    <name-node>\$(nameNode)</name-node>
    <configuration>
      <property>
        <name>mapred.mapper.class</name>
        <value>uk.ac.ic.doc.dron.WordMapper</value>
      </property>
      <property>
        <name>mapred.reducer.class</name>
        <value>uk.ac.ic.doc.dron.WordReducer</value>
      </property>
      <property>
        <name>mapred.map.tasks</name>
        <value>1</value>
      </property>
      <property>
        <name>mapred.intput.dir</name>
        <value>document</value>
      </property>
      <property>
        <name>mapred.output.dir</name>
        <value>word_count</value>
      </property>
    </configuration>
  </map-reduce>
  <ok to="UniqueWords">
  <error to="fail">
</action>
```

Listing 2.5: Example of an Oozie MapReduce Job

```
<action name="UniqueWords">
  <pig>
    <job-tracker>\$(jobTracker)</job-tracker>
    <name-node>\$(nameNode)</name-node>
    <script>uk/ac/ic/doc/dron/unique.pig</script>
    <param>INPUT=word_count</param>
    <param>OUTPUT=unique_words</param>
  </pig>
  <ok to="end">
  <error to="fail">
</action>
```

Listing 2.6: Example of an Oozie Pig Job

Having evaluated Oozie, we believe that it is a system that meets a significant amount of our requirements. However, we would like to point several of its shortcomings:

1. It has been designed to only work with the Hadoop framework. Moreover, its architecture makes it difficult to integrate any new system. The lack of support for HBase and Hive serves as evidence.

2. The requirement of expressing workflows in XML files makes the scheduler difficult to use. The workflows can quickly become difficult to manage. As we could see in the example we presented, files grow to a considerable size even for the most simple jobs. Thus, we forsee the need of tools that reduce the complexity of Oozie workflow creation and management.

3. Oozie is not suitable for multi-user enviroments. We believe that our users workflows have common steps. Thus, we emphasize the importance of allowing one user to depend on another user's job.

Finally, we would like to mention that Oozie has encouraged the developement of another new system called Nova[39]. It is a workflow manager that lays on top of Oozie and it provides continuous Pig/Hadoop workflows. However, we will not go into further details because Nova has not tried to solve what we believe are Oozie's limitations.

2.7.3 Condor and DAGMan

Condor[40] is the most popular high-throughput distributed batch computing system. Its goal is to provide large amounts of computational power while effectively utilizing the resources available to the network. Thus, Condor supports resource management and monitoring, scheduling policies and job management.

As a result of its long withstanding existence and success, many tools have been built on top of it. **DAGMan (Directed Acyclic Graph Manager)** is just one of them. It provides a solution for running jobs with dependencies. Figure 2.5 exemplifies a workflow that can be expressed with the help of DAGMan. Moreover, Listing 2.7 gives the language definition of the graph. Jobs are expressed with the JOB statement that associates names with files containing Condor job descriptions. The directed edges of the graph are expressed with the help of the PARENT-CHILD statement that describes the dependency of two jobs. An interesting feature is the support of PRE and POST scripts that can be run before and respectively after the execution of a job.

```
1 JOB ScrapeData sd.condor
2 JOB AnalyzeData1 ad1.condor
3 JOB FilterData fd.condor
4 JOB AnalyzeData2 ad2.condor
5 PARENT ScrapeData CHILD AnalyzeData1 FilterData
6 PARENT FilterData CHILD AnalyzeData2
7 SCRIPT PRE FilterData in.pl
8 SCRIPT POST FilterData out.pl
```

Listing 2.7: Example of DAGMan's supported language

DAGMan has been designed to gracefully handle job failures. For example, if a job terminates with a result indicating an error, the system writes out a rescue DAG. The new graph contains only the jobs that have not been executed before the failure.

Much research has been conducted around Condor and DAGMan. For example, the authors of [41] have succeeded to optimize the run of a dependencies graph by trying to

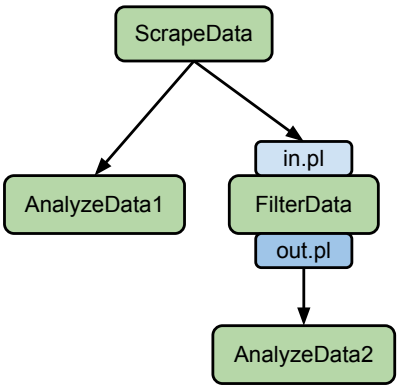

Figure 2.5: Example of a DAGMan Job Graph

determine an Internet-Computing optimal (IC optimal) schedule. The goal of such a schedule is to devise an order for assigning jobs to workers such that the number of jobs that can be run at every step is maximized. The tool implemented by the authors has managed to reduce the execution time by 13%.

While Condor and DAGMan have long proved to be able to handle many different types of jobs, we believe that they have several shortcomings. For example, running MapReduce jobs requires an active client process to be kept open. Thus, the potential Condor job that runs a MapReduce computation, may terminate before the actual work is over. As a result, DAGMan will start running the jobs that depend on the MapReduce results.

Lastly, another potentially major drawback is the fact that the systems were not designed to support computation sharing between users. As each dependencies DAG is defined within a file, there is no central knowledge of a complete DAG of jobs. As a result, users may end up performing the same computation more than once. For example, a user may not be aware of an already existing job (e.g. daily production database scaper) and may proceed into duplicating the work by writing his own new job.

2.7.4 Gearman

Gearman[31] is a framework that can be used to farm out work to other machines. It currently supports two types of jobs: foregroung and background. The former are jobs that have a client attached while the latter do not. Figure 2.6 depicts the main components of the system: job managers, workers and clients.

The clients are responsible of creating jobs and requesting the job server to run them. Whenever the server receives a request, it dispatches it to one of the managed workers. Following, the worker performs the required work and sends a response to the client through the job server.

The job server stores all the jobs in memory based queues. Thus, if the server fails while having pending jobs, they will be lost forever. Gearman partially solves the problem by

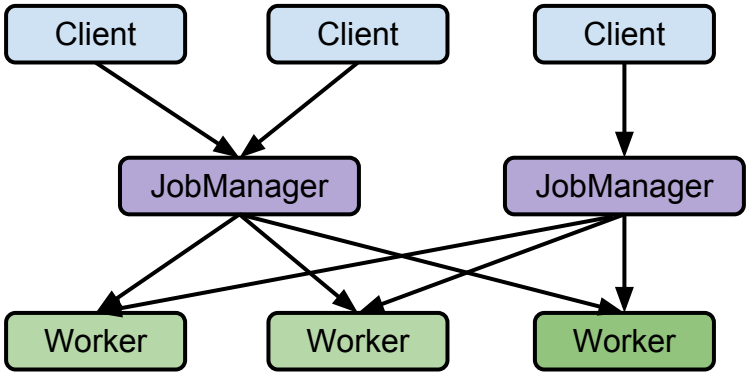

Figure 2.6: Gearman Overview

supporting persistent queues. The pending jobs are written into a database so that they survive server restarts and failures. However, the approach does not provide availability because in the case of a hardware failure the framework will not be able to run the pending jobs.

While Gearman has met the demands of many companies, we believe that it does not provide the abstractions required to run complex workflows on it. In its current state it does not support any way of expressing dependencies between jobs. Thus, a user would have to implement his own mechanism of checking dependencies. Moreover, background job failures can cause headaches to the users, as there is no current mechanism for detecting them.

To sum up, we think that German would have to go through a complete redesign in order to be able to implicitly support many frameworks, job dependencies and nonetheless improve its fault tolerance.

2.7.5 Cascading

Cascading[28] is a data processing API and process scheduler. It can be used to execute data processing workflows on a cluster of machines. Cascading has been built to reduce the complexity of developing applications on top of the Apache Hadoop framework.

The data processing workflows are build using the Cascading API. The interface introduces several new concepts such as *pipe assemblies* and *operations*. A *pipe assembly* is a pipeline that may or not have data sources and data sinks. There currently are five types of pipe assemblies that apply different operations to the content of the streams (i.e. tuples).

- **Each** - it applies a function or a filter operation to every tuple from the stream.
- **GroupBy** - it groups the stream on selected fields from the tuple.
- **CoGroup** - similar to the SQL join operation, it joins streams based on some common set of fields.

- **Every** - applies an aggregator function to every tuple from the stream.
- **SubAssembly** - packages pipe assemblies to be used in a larger pipe assembly.

Cascading supports custom operations by allowing the users to define their own functions. There are four types of functions: *Function*, *Filter*, *Aggregator* and *Buffer*. The first one expects a single *Tuple* as argument and it may return zero or more *Tuples* as result. The *Filter*, as its name suggests, expects a single *Tuple* as argument and returns a boolean value stating if the tuple should be discarded from the stream or not. The third operation expects a set of *Tuples* in the same grouping as input and returns one or more *Tuples*. Lastly, the *Buffer* is similar to the *Aggregator*, it only differs in its input. It receives the current *Grouping Tuple*.

After defining pipe assemblies and their operations, one has to bind them to sources and sinks. The process of binding results in a *Flow*. Subsequently, *Flows* can be grouped together to form a *Cascade*. The grouping is done via dependencies (i.e. a *Flow* can depend on the output of another *Flow*).

Finally, the defined Cascade job is passed to Cascading's MapReduce Job Planner. Its role is to convert the pipe assemblies into a graph of dependant MapReduce jobs. Furthermore, Cascading has a scheduler that given a graph of dependant flows runs them on the Hadoop framework according to their topological order.

Having presented Cascading's features we would like to point several of its limitations. Firstly, it does not provide failure guarantees for long running jobs. Since Cascading is running from the client's machine, there is no way to provide availability and dataflow recoverability in case of hardware failures. Secondly, we believe that the API which is provided may be a bit restrictive. For example, one would not be able to easily express jobs that do not involve stream operations. Lastly, its architecture does not provide a way of adding non MapReduce based computation frameworks. In other words Cascading it is not able to run more complex computations supported by CIEL or Dryad.

2.7.6 Quartz

Quartz[32] is an open source job scheduler that has been designed to be used together with any Java application. Users can specify the time when a job should be executed by providing Cron style strings. Quartz transforms the strings into triggers that notify the scheduler when a job must be run.

Similar to Gearman, the system can persist the jobs or it can optimize by only storing them into the memory. In contrast to the abovementioned system, Quartz does provide fault tolerance by being able to run a cluster of schedulers. Whener a machhine fails, one of the remaining schedulers reads the pending jobs from the database and runs them.

Quartz can send scheduling/job events to applications that implement several of the available listeners (e.g. JobListener, TriggerListener). Thus, even tough the scheduler does not natively support dependencies between jobs, one could programmatically create workflows by launching the dependant jobs when completion events are received. The solution we provided may be good enough for simple cases, but for the following reasons it is not suitable for expressing long running complex workflows:

- Every single user will have to manually reimplement dependency checking in order to be able to express his or hers workflows.
- It requires the client to be connected to the scheduler for the whole duration of the computation. Hence, it introduces a new point of failure in the system.
- Users are required to explicitly handle job failures in their programatic workflows.
- Applications will not be aware of the computation each one is conducting. Hence, they may end up running the same jobs more than once.

To sum up, we believe that Quartz is a great scheduler for simple Java based jobs. However, since it has been so tightly coupled with the language, it does not provide any way of integrating external frameworks.

2.7.7 Moab

Adaptive Computing has developed a suite of HPC workload management products. Among them, a scheduling and management system called Moab Workload Management [33]. The product was built to centralize the access to clusters and to optimize resource usage. As a result the system supports four types of workloads:

- Batch - tipically a job command file that describes all the jobs requirements (e.g. memory, disk, cpu). When the job is submitted it is placed into a job queue. Subseqnetly, when resources become available, it run in the cluster.
- Interactive - a job whose output is of immediate interested to the users. Thus, the job submitters remain connected to the job so that users can view output and error information in real-time.
- Calendar - jobs that must be executed at a particular time or on regular basis.
- Service - long-running jobs that usually are persistent services. Moab supports extensive configuration for this type of jobs. As an example a user may specify storage requirements and locality constraints.

The aforementioned jobs pass through a series of states as they are scheduled. The Table 2.3 briefly explains the most important ones.

Sadly, we were not able to find any document explaining the architecture of the product. However, in the Administrator Guide, it was briefly mentioned how the system achieves high availability. The scheduler has several slave instances running such that they can take over scheduling in case of a failure.

While evaluating the product we were impressed by its abilities to handle jobs with priorities and by its extensive resource management options. However, we believe that the system is not suitable for the types of computation we are trying to run as it has not been designed for clusters of commodities machines. Moreover, its complexity makes it very difficult to try to adapt it to support the current state of the art frameworks. The above mentioned

State	Description
deferred	the job has been postponed due to an inability to schedule it
idle	the job is queued and is waiting to be run
starting	the job is performing pre-start tasks (e.g. provisiong resources)
running	job is currently executing
canceling	job has been canceled
completed	job has successfully completed

Table 2.3: Job States

arguments together with the fact that it does not support job dependencies have determined us to rule it out as a solution for our problem.

2.8 Summary

In this section we provided an overview of the current state of the art frameworks designed for large computation and data analysis. Following, we have studied five job schedulers and two workflow management systems.

Project Plan

At the moment I have managed to build the core of the scheduler. However, while testing the system I have found several limitations that do not allow it to scale. As a result my plan for the rest of the project is the following:

- **15th of January - 3rd of February**
  The tests that I have currently conducted have revealed several bottlenecks. Thus, I plan to redesign the system over the following three weeks in order to improve its scalability.
- **4th of February - 8th of March**
  I plan to develop the bulk of the dependencies language that Dron is going to support.
- **9th of March - 24th of March**
  The two weeks will be spent on testing the fault-tolerance and the scalability of the system under the loads described in the following chapter.
- **25th of March - 2nd of April**
  I plan to adapt the remaining systems and to build a recommendation engine with the help of Dron.
- **15th of May - 30th of May**
  Dependening on how well the work progressed up to this point, I plan to add a simple resource manager to the scheduler. This may open the door to a complete new set of scheduling experiments.
- **31st of May - 15th of June**
  The last two to three weeks will be spent on finishing writing up the report.

I think that the biggest risk of the project is that developing a system such as Dron is very time consuming. Moreover, up to now I have continuosly ran into scalability and fault tolerance issues. Thus, in the eventuality I will not be able to make the system scalable, I will concentrate more on the dependencies language.

Evaluation Plan

I believe that project evaluation is very important. Thus, I will try to use several tests that stress out various features. The evaluation I will be conducting will be of two types. The first type will try to push the limits of the system. I will deploy it on approximatively 100 Amazon nodes and I will run the following tests:

1. **Short Duration Jobs** - I plan to test the performance of the scheduler by running as many as possible short jobs (e.g. a simple echo). These jobs will show us which are the bottlenecks of the system and its upper limit in terms of jobs per second.

2. **Long Jobs** - Since short jobs do not test the entire scalability of the system, I plan to evaluate how the system behaves while handling many long duration jobs. They will stress the system's ability to handle many timers at a specific time.

3. **Facebook Load** - My Facebook contacts have agreed to provide me a description of the jobs they are running on their internal job scheduler. Thus, I will test Dron's suitability in handling the load that their system is supporting.

4. **Google's Job Patterns** - Google has a program by which students/researchers can request access to information about Google workloads. I plan to use this information to create a set of jobs to be run on Dron.

5. **Oozie** - I plan to compare the scalability of Dron with Oozie's. Thus, I will run several tests composed of small and long duration jobs. Finally, I will try to compare the system by running several MapReduce jobs.

An important note to make is that I am testing the system as I am developing it. This has helped me to already detect several bottlenecks. For example a Mnesia node can not handle more than 1400 transactions per second. This has forced me to change the design of the scheduler in order to make it more scalable.

The second type of evaluation I plan to run will be designed to analyze the expesivity of Dron's workflows and to test the scheduler's usage simplicity.

1. **Integrate Several Frameworks** - I plan to demonstrate the easyness of adapting new frameworks with Dron. I have already partially completed this step by integrating Hadoop MapReduce, Spark and Mahout. In addition to the beforementioned I will adapt HDFS and Hive.

2. **Build a Recommendation Service** - I will be building a small recommendation engine based on Mahout, MapReduce and Dron. The example should demonstrate the power of Dron's dependencies language.

The following chapters represent a work in progress.

Chapter 3

Dron Overview

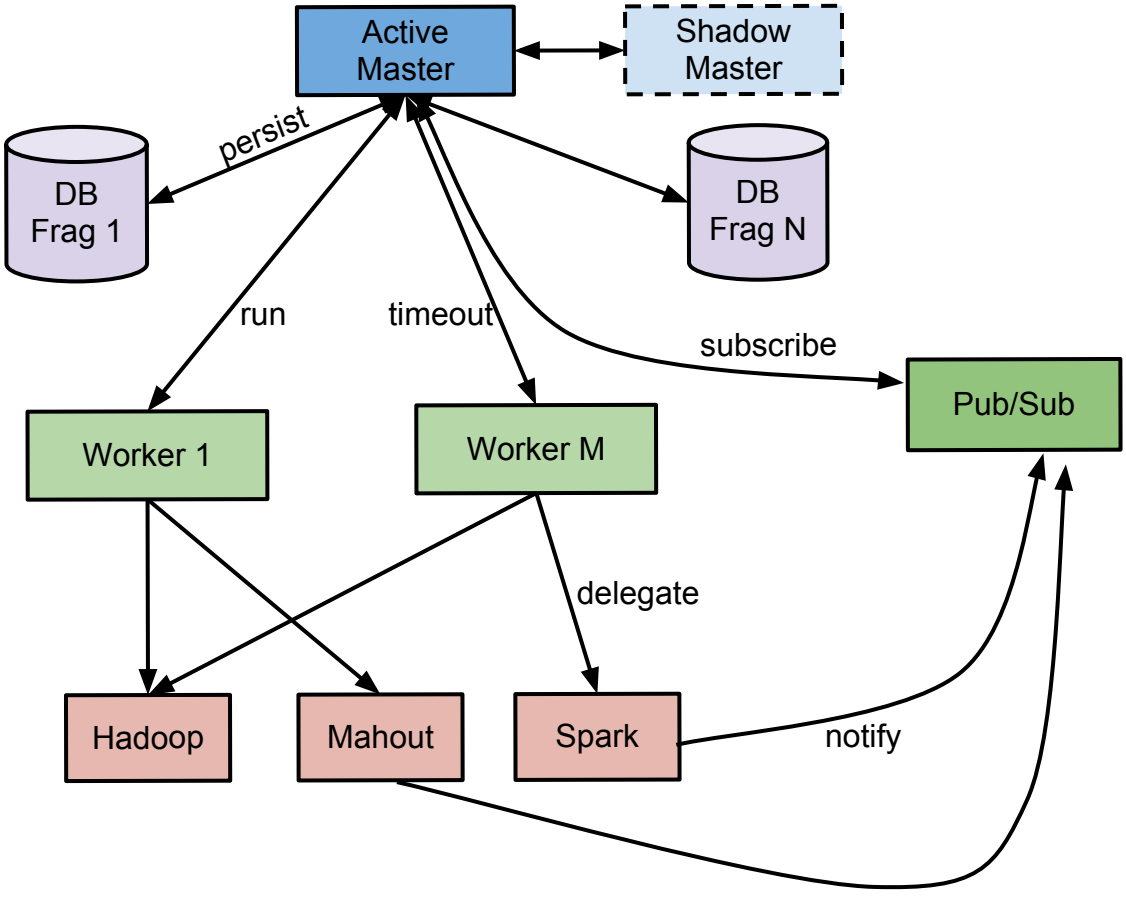

Figure 3.1: Architecture Overview

Chapter 4

Design Choices and Implementation

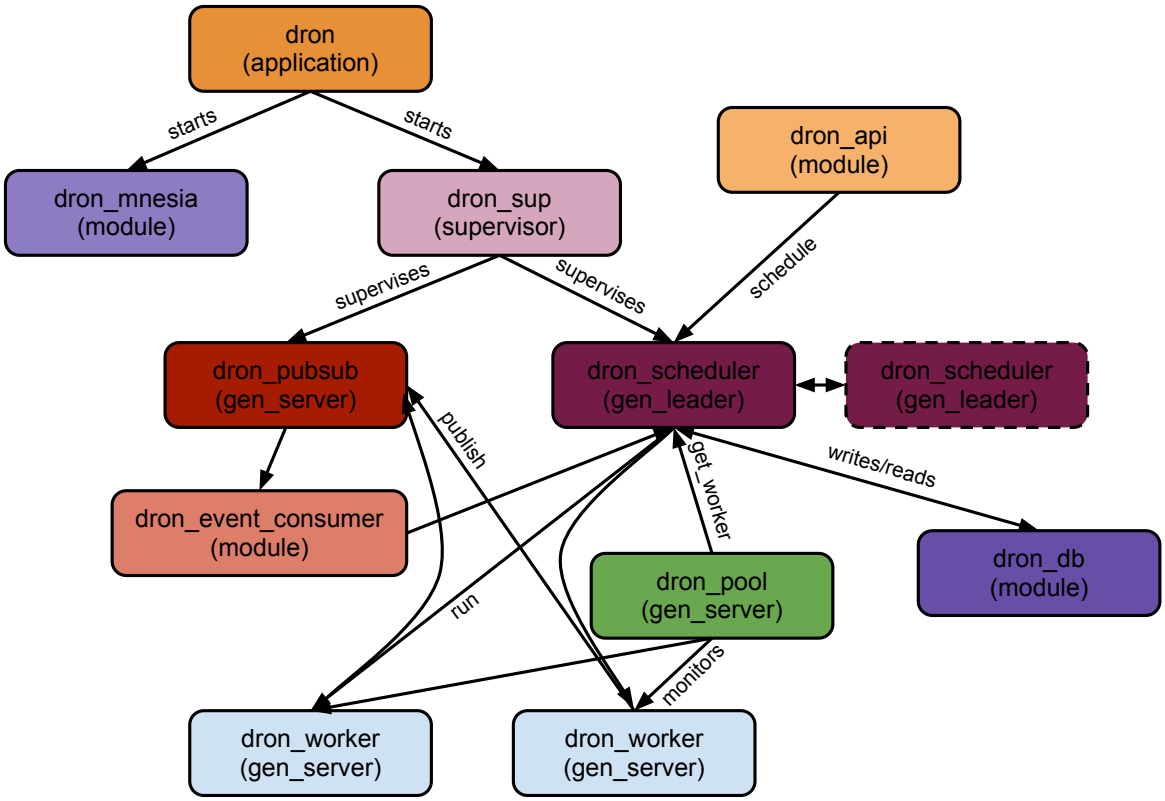

Figure 4.1: Implementation Overview

Chapter 5

Integrating Frameworks

5.1 Introduction

An important goal of the system was to provide simple ways of integrating frameworks within the scheduler.

When a developer wants to integrate a new framework he or she will have to send notification events for every job completion/failure. The events will be sent using an implementation of AMQP (RabbitMQ). Following, depending on the choice the developer makes he or she will have two options:

1. Change the framework to support job parameters. Thus, everytime when a Dron runs a job instance it will pass the dron job instance id to the framework. This approach will allow the developer to send events on a common queue (dron_events) shared by Dron and all the frameworks.

2. Provide a module that filters events from a specific framework. Subsequently, it transforms the events into dron understandable information

5.2 Hadoop

5.2.1 MapReduce

5.2.2 Hadoop Distributed File System

5.3 Spark

Knowking Hadoop's weakness in running iterative jobs, I have decided to integrate the Spark[16] cluster computing system as well. The Scala based framework provides primitives for in-memory cluster computing. Thus, iterative jobs can quickly access the data instead of writing and reading it from the disk as in Hadoop.

Being confrunted with a relatively small project I have decided to adapt it such that it can receive Dron job instance id whenever a new Spark job is run. This has been achieved by adding the knowledge to the two classes modelling Spark jobs. Subsequently, I had to provide a RabbitMQ publisher for Spark. Even tough there is no RabbitMQ client library for Scala I could easily reuse the Java implementation. Lastly, I had to adjust the Spark scheduler to send job failure/success events to Dron. We expect that the performance of the framework was not affected as the code that was added in the core of the system is not computation intensive. Listing 5.1 provides an extract of it. However, if a performance dropped will be observed later on, one can improve the extension by adding a pool of event dispatcher threads that adjusts its size according to the number of elements in producer consumer queue.

```
1 if (job.hasFailed() == true) {
2   val jsonMap: Map[String, Any] = Map("job_instance" -> dronInstanceId,
3                                       "state" -> "failed")
4   val jsonObject = new JSONObject(jsonMap.toMap)
5   dron.publishMessage(jsonObject.toString())
6 } else {
7   val jsonMap: Map[String, Any] = Map("job_instance" -> dronInstanceId,
8                                       "state" -> "succeeded")
9   val jsonObject = new JSONObject(jsonMap.toMap)
10  dron.publishMessage(jsonObject.toString())
11 }
```

Listing 5.1: Additions to the Mesos scheduler: whenever a job completes, a JSON object is constructed and published according to the job's state.

5.4 Mahout

Mahout uses a driver program to run machine learning computations on Hadoop. By having a clear entry point to a Mahout job, I could easily adapt the library. The driver receives the address of the RabbitMQ host, the name of the exchange and the Dron job instance id correspoding to the Mahout job as command line arguments.

```
1 if (host != null && exchange != null && dronJobInstanceId != null) {
2   Dron dron = new Dron(host);
3   dron.publish(exchange, dron.buildJsonString(dronJobInstanceId, status));
4   dron.close();
5 }
```

5.5 Hive

Chapter 6

Dron's Fault Tolerance

Chapter 7

Recommendation Engine Based on Dron and Mahout

Chapter 8

Evaluation

8.1 Latency

8.2 High Load

8.3 Facebook Load

Chapter 9

Conclusions

Bibliography

[1] Gantz, J., Reinsel, D. *Extracted Value from Chaos*

[2] Dean, J. *https://sites.google.com/site/io/underneath-the-covers-at-google-current-systems-and-future-directions*

[3] Dean, J., Ghemawat, S. *MapReduce: Simplified Data Processing on Large Clusters.* Communications of ACM Volume 51, 1 (Jan. 2008).

[4] Hadoop. *http://hadoop.apache.org.*

[5] Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D. *Dryad: Distributed Data-parallel Programs from Sequential Building Blocks.* In Proceedings of the 2nd European Conference on Computer Systems (Eurosys 2007).

[6] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P., Currey, J. *DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.* Symposium on Operating System Design and Implementation (OSDI, Dec. 2008).

[7] Thusso, A., Sarma, J., Namit, J., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P. and Murthy, R. *Hive: A Warehousing Solution Over a Map-reduce Framework.* In Proceedings of the VLDB Endowment.

[8] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A. *Pig latin: a not-so-foreign language for data processing* In Proceedings

of the ACM SIGMOID International Conference on Management of Data (2008).

[9] Pike, R., Dorward, S., Griesemer, R., Quinlan, S. *Interpreting the Data: Parallel Analysis with Sawzall.* Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure (2005).

[10] Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R., Bradshaw, R., Weiznbaum, N. *FlumeJava: Easy, Efficient Data-parallel Pipelines.* In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.

[11] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sarma, J., Murthy, R., Liu, H. *Data Warehousing and Analytics Infrastructure at Facebook.* In Proceedings of ACM SIGMOD 2010 Conference on Management of Data.

[12] Scribe. *https://github.com/facebook/scribe.*

[13] Hadoop Distributed File System. *http://hadoop.apache.org/hdfs.*

[14] Page, L., Brin, S., Motwani, R., Winograd, T. *The PageRank Citation Ranking: Bringing Order to the Web.* In Technical Report, Stanford InfoLab (1999).

[15] Apache Mahout. *http://mahout.apache.org.*

[16] Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I. *Spark: cluster computing with working sets.* In Proceddings of the 7th USENIX Conference on Networked Systems Design and Implementation (2010).

[17] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, J., Shenker, S., Stoica, I. *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing.* Technical Report UCB/EECS-2011-08.

[18] Bu, Y., Howe, B., Balazinska, M., Ernst, M. *HaLoop: Efficient Iterative Data Processing On Large Clusters.* In Proceedings of the VLDB Endowment (2010).

[19] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S., Qiu, J., Fox, G. *Twister: A Runtime For Iterative MapReduce.* In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010).

[20] Murray, D., Hand, S. *Scripting the Cloud with Skywriting.* In Proceedings of the 2nd USENIX Workshop on Hot Topics (Jun. 2010).

[21] Murray, D., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., Hand, S. *CIEL: A Universal Execution Engine for Distributed Data-flow Computing.* In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (2011).

[22] Malewicz, G., Austern, M., Bik, A., Dehnert, J., Horn, I., Leiser, N., Czajkowski, G. *Pregel: A System for Large-scale Graph Processing.* In Proceedings of the ACM SIGMOID International Conference on Management of Data (2010).

[23] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A., Katz, R., Shenker, S., Stoica, I. *Mesos: A Platform for Fine-grained Resource Sharing in the Data Center.* In Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (2011).

[24] Hindman, B., Konwinski, M., Zaharia, M., Stoica, I. *A Common Substrate for Cluster Computing.* In Proceedings of the 2009 Conference on Hot Topics in Cloud Computing (2009).

[25] Hunt, P., Konar, M., Junqueira, F., Reed, B. *ZooKeeper: wait-free coordination for internet-scale systems.* In Proceedings of the USENIX Annual Technical Conference (2010).

[26] Karger, D., Lehman, D., Leighton, T., Panigrahy, R., Levine, M., Lewin, D. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.* In Proceedings of STOC 1977 Symposium on Theory of Computing.

[27] Ghemawat, S., Gobioff, H., Leung, S. *The Google File System.* In 19th ACM Symposium on Operating Systems Priciples (2003).

[28] Cascading. *http://www.cascading.org.*

[29] Power, R., Li, J. *Piccolo: Building Fast, Distributed Programs with Partitioned Tables.* In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (2010).

[30] Oozie. *http://yahoo.github.com/oozie.*

[31] Gearman. *http://gearman.org.*

[32] Quartz. *http://www.quartz-scheduler.org.*

[33] Moab. *http://www.clusterresources.com.*

[34] Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., Wong, M. *Tenzing: A SQL Implementation On The MapReduce Framework* In Proceedings of the VLDB Endowment Vol. 4 (2011).

[35] Melnik, S., Gubarev, A., Ling, J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T. *Dremel: Interactive Analysis of Web-Scale Datasets* In Proceedings of the 36th VLDB Conference (2010).

[36] Thirft. *http://thrift.apache.org/*

[37] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R. *Bigtable: A Distributed Storage System for Structured Data* In ACM Transactions on Computer Systems (June 2008).

[38] Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J., Sears, R. *BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud* In Proceedings of the 5th Eurosys Conference (2010).

[39] Olston, C., Chiou, G., Chitnis, L., Liu, F., Han, Y., Larsson, M., Neumann, A., Rao, V., Sankarasubramanian, V., Seth, S., Tian, C., ZiCornell, T., Wang, X. *Nova: Continuous Pig/Hadoop Workflows* In Proceedings of SIGMOD Conference (2011).

[40] Thain, D., Tannenbaum, T., Livny, M. *Distributed Computing in Practice: the Condor Experience* Concurrency and Computation - Practice and Experience

[41] Malewicz, G., Foster, I., Rosenberg, A., Wilde, M. *A Tool for Prioritizing DAGMan Jobs and its Evaluation* In Journal of Grid Computing Volume 5 (2007).

[42] Peng, D., Dabek, F. *Large-scale Incremental Processing Using Distributed Transactions and Notifications* In Proceedings of 9th USENIX Symposium of Operating Systems Design and Implementation (2010).

[43] Popa, L., Budiu, M., Yu, Y., Isard, M. *DryadInc: Reusing Work in Large-scale Computations* In Proceedings of HotCloud Conference on Hot Topics in Cloud Computing (2009).