Programme ANR VERSO

Projet VIPEER

Ingénierie du trafic vidéo en intradomaine basée sur les paradigmes du Pair à Pair

Décision nº 2009 VERSO 014 01 à 06 du 22 décembre 2009
T0 administratif = 15 Novembre 2009
T0 technique = 1er Janvier 2010

Deliverable 4.4

Report on CDN/dCDN modeling and analysis

Auteurs:
C. Bothorel, R. Picot-Clémente, Z. Li (Telecom Bretagne),
P. Michiardi (Eurocom),
Y. Hadjadj-Aoul (INRIA),
J. Garnier (NDS Technologies France)

Edited by:
J. Garnier (NDS Technologies France)

June 2012

Telecom Bretagne; Eurocom; INRIA; NDS Technologies

Abstract

This document aims to present the report on CDN/dCDN modeling and analysis based on the previous deliverable. On the previous deliverable, we introduced the Parallel Processing as it is a key point for our architecture. So this deliverable will first present an update regarding the MapReduce framework. As for now, this framework is well integrated in the WP4 but another key point is to see how to co-locate Hadoop clusters in order to perform a multitude of analytics jobs. Then, regarding the prefetching, we want to go more deeply in the algorithms and parameters for having a better recommendation engine. The output of this engine will still be used as an input for the genetic algorithm (GA). We will present the integration of the MapReduce framework for the GA. Finally, we will expose our approach regarding the distributed replication and caching strategy for ViPeeR.

Keywords: CCN, Prefetching, MapReduce

Contents

1 Preface

1.1 Purpose of this document

This document aims to present the report on CDN/dCDN modeling and analysis based on the previous deliverable. On the previous deliverable, we introduced the Parallel Processing as it is a key point for our architecture. So this deliverable will first present an update regarding the MapReduce framework. As for now, this framework is well integrated in the WP4 but another key point is to see how to co-locate Hadoop clusters in order to perform a multitude of analytics jobs. Then, regarding the prefetching, we want to go more deeply in the algorithms and parameters for having a better recommendation engine. The output of this engine will still be used as an input for the genetic algorithm (GA). We will present the integration of the MapReduce framework for the GA. Finally, we will expose our approach regarding the distributed replication and caching strategy for ViPeeR.

1.2 Referenced ViPeeR deliverables

Table 1 lists documents and other reference sources containing information that may be essential to understanding topics in this document.

No	Designation	Title
1.	D4.1	State of the Art
2.	D4.2	Preliminary report on the CDN/dCDN design
3.	D4.3	Report on the CDN/dCDN design

2 Parallel Processing in ViPeeR

In this Chapter we describe the research activities on the Hadoop parallel processing framework used in ViPeeR. The following sections explain our work on a novel scheduling algorithm that targets the problem of co-location of Hadoop clusters, when they are used to perform a multitude of analytics jobs.

2.1 Introduction

The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce [18] and Dryad [23], has created the need to organize and manage the resources of clusters of computers that operate in a shared, multi-tenant environment. Initially designed for few and very specific batch processing jobs, data-intensive scalable computing frameworks are nowadays used by many companies (e.g. Twitter, Facebook, LinkedIn, Google, Yahoo!, ...) for production, recurrent and even experimental data analysis jobs. Within the same company, many users *share* the same cluster because this avoids redundancy (both in physical deployments and in data storage) and may represent enormous cost savings.

In this work, we study the problem of resource *scheduling*, that is how to allocate the (computational) resources of a cluster to a number of concurrent jobs submitted by the users, and focus on the open-source implementation of MapReduce, namely Hadoop [2]. Despite scheduling is a well known research domain, the distributed nature of data-intensive scalable computing frameworks makes it particularly challenging. In addition to the default, first-in-first-out (FIFO) scheduler implemented in Hadoop, recently, several solutions to the problem have been proposed to the community [37, 9, 20, 26, 31, 36]: in general, existing approaches aim at two key objectives, namely *fairness* and *performance*.

Given the state-of-the-art, it is natural to question the need for another approach to scheduling cluster resources. In this work we observe that fairness and performance are non-conflicting goals, hence there is no reason to focus solely on one or the other objectives. We thus proceed with the design of a scheduling protocol that can be implemented in practice, and that caters *both* to a fair *and* efficient utilization of a shared cluster. Our solution, called Hadoop Fair Sojourn Protocol (HFSP), is inspired by FSP [19], and represents an extension of FSP from the single server case to the multiple server case.

HFSP belongs to the category of the size-based, preemptive scheduling disciplines, therefore it requires the knowledge of the duration of the jobs. We address this by estimating the job size with a training set, paying attention to not waste

resources or the job done during the estimation process. Moreover, we need to introduce the preemption, so that small jobs do not have to wait behind long jobs.

Scheduling in a multi-server environment is not trivial, since the scheduling algorithm needs to account for the discrete nature of slots available to execute parallel jobs, and MapReduce job has a complex structure, composed by Map and Reduce phases.

The contribution of ou work can be summarized as follows:

- We design and implement different building blocks that are used by the scheduling discipline: in particular we provide a training queue where the job size can be estimated, and a set of primitives for the implementation of the preemption. Moreover we implement a dynamic assignment of the resources to the training queue in order to avoid cluster under-utilization.

- We design and implement a multi-server scheduling discipline that is able to provide a processor-sharing-like fairness among jobs, and, at the same time, tries to minimize the execution time.

- We perform an extensive experiment campaign, where we copare our HFSP scheduler with the to the two main schedulers used in production today. For the experiments, we use state of the art workload suite generators that take as input realistic workload traces as input, and contribute to their further development. The results shows that the execution time is drastically reduced. Moreover, we are able to analyze where and why other schedulers underperform, under a variety of workloads.

The remainder of the Chapter is organized as follows: In Sect. 2.2 we provide some background on the scheduling disciplines and on MapReduce. In Sect. 2.3 we describe the design of the different components that forms our solution. We evaluate the performance of our job scheduler in Sect. 2.4. We provide in Sect. 2.5 additional consideration. In Sect. 2.6 we discuss the related work, and we conclude Sect. 2.7.

2.2 Background

Scheduling disciplines have been widely studied in the past, especially in the context of computer networks [21]. In this section we give the necessary background to understand the main idea we develop in this work.

When comparing different scheduling disciplines, there are different performance indexes. We consider the main two indexes, namely (i) the *mean response time* – *i.e.* the total time spent in the system, given by the waiting and service time – for each job, and (ii) the *fairness* across jobs – in particular, we consider the notion of fairness as equal share of the system resources.

Among all the scheduling disciplines proposed in the literature (for a general overview, see [21] and the references therein), we focus on two policies that are relevant in our context: a policy that minimizes the mean response time and one that provides perfect fairness.

The optimal pre-emptive scheduling policy that minimize the mean response time is the Shortest Remaining Processing Time (SRPT), where the job in service is the one with the smallest remaining processing time – this policy requires the job size to be known *a priori*. SRPT focuses on the mean response time, while it provides no guarantees on system fairness: as such, long jobs may starve.

As opposed to minimizing the mean response time, the Processor Sharing (PS) discipline is conceived to guarantee a fair share of system resources to be dedicated to each job: if N jobs need to be served, with PS each receives a $1/N$th fraction of the system resources. However, the mean response time achieved by PS is higher than that obtained with SRPT.

In [19], the authors provide a scheduling policy (which requires the job size) that strives to obtain both (near) optimal mean response times for all jobs and fairness across all jobs, called Fair Sojourn Protocol (FSP). Since our work builds upon FSP, in the following we provide sufficient background to understand its properties.

2.2.1 How FSP Works

The main idea of FSP is to run jobs in series rather than in parallel. In practice, assuming a PS policy, where each job has its fair share, it is possible to compute the completion time for each job under this discipline. The order at which jobs complete in PS is used by FSP as a reference to schedule jobs in series. In the basic single server configuration, this means that at most one job is served at a time, and that such job may be preempted by a newly arrived job. An example is the best way to illustrate how FSP works.

Assume that there are three jobs, j_1, j_2 and j_3, each requiring all the resources available in the system. Such jobs arrive at time $t_1 = 0$s, $t_2 = 10$s and $t_3 = 15$s respectively; it takes 30 seconds to process job j_1, 10 seconds to process job j_2 and 10 seconds to process job j_3 (if all the resources are used, otherwise the time increases inversely proportionally to the available resources).

Figure 2.1 (top) represents the system utilization over time under the PS discipline: when job j_2 arrives, the server is shared between j_1 and j_2, and, when job j_3 arrives, the server is shared among the three jobs. The job completion order is j_2, j_3 and j_1. The bottom part of the figure shows how the workload described above is scheduled under the FSP discipline. When job j_2 arrives, since it would finish before job j_1 in case of PS, it preempts job j_1. When job j_3 arrives, it does not preempt job j_2, since it would finish after it in case of PS; when job j_2 finishes, job j_3 is scheduled since it would finish before job j_1 in case of PS.

The FSP scheme ensures each job to receive a fair amount of system resources, as when PS scheduling is used. At the same time, under FSP, the mean job completion time is considerably smaller than under PS. In particular, long jobs tends to have the same completion time as in PS, while short jobs finish before.

While the definition of FSP, in case of a single server or in case of jobs that always require all the available resources, is simple, when we consider multiple servers with jobs that may require less than 100% of the system resources, we need to manage different cases. Next, using a simple example, we anticipate a more elaborate setup

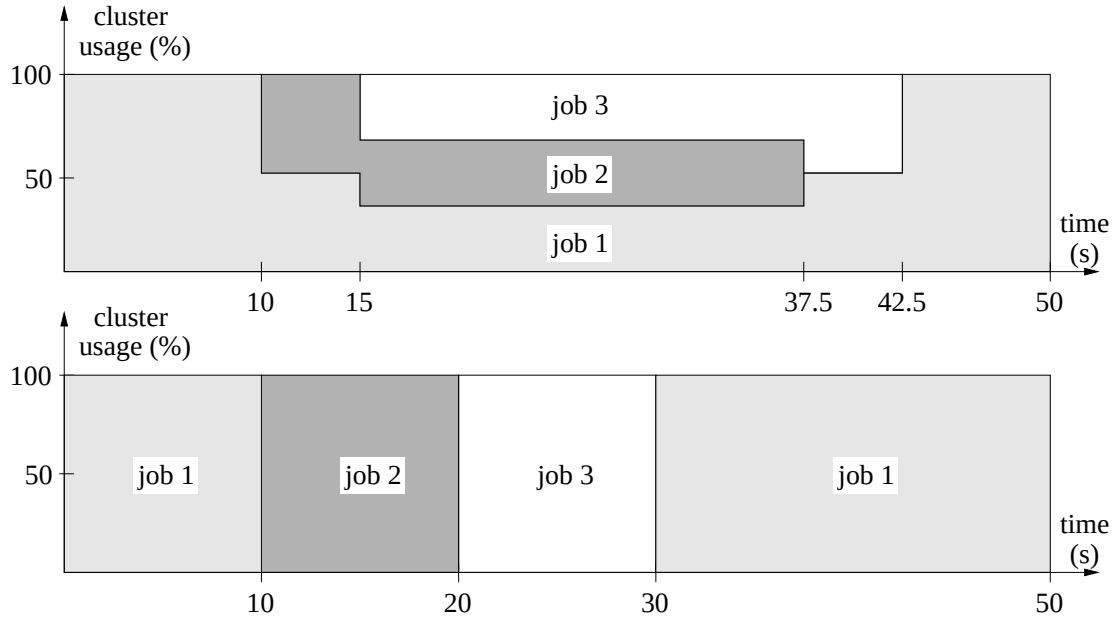


Figure 2.1: Comparison between PS (top) and FSP (bottom).

that underlies our work, whereas in Sect. 2.3.5 we detail all the hidden intricacies of a parallel version of FSP, called Hadoop Fair Sojourn Protocol (HFSP).

Assume that jobs j_1, j_2 and j_3 require 100%, 55% and 35% of the system resources respectively. The arrival times are $t_1 = 0$s, $t_2 = 10$s and $t_3 = 13$s and the processing time (if the required share of system resources is given to each job) is 30 seconds for job j_1, 10 seconds for job j_2 and 10 seconds for job j_3.

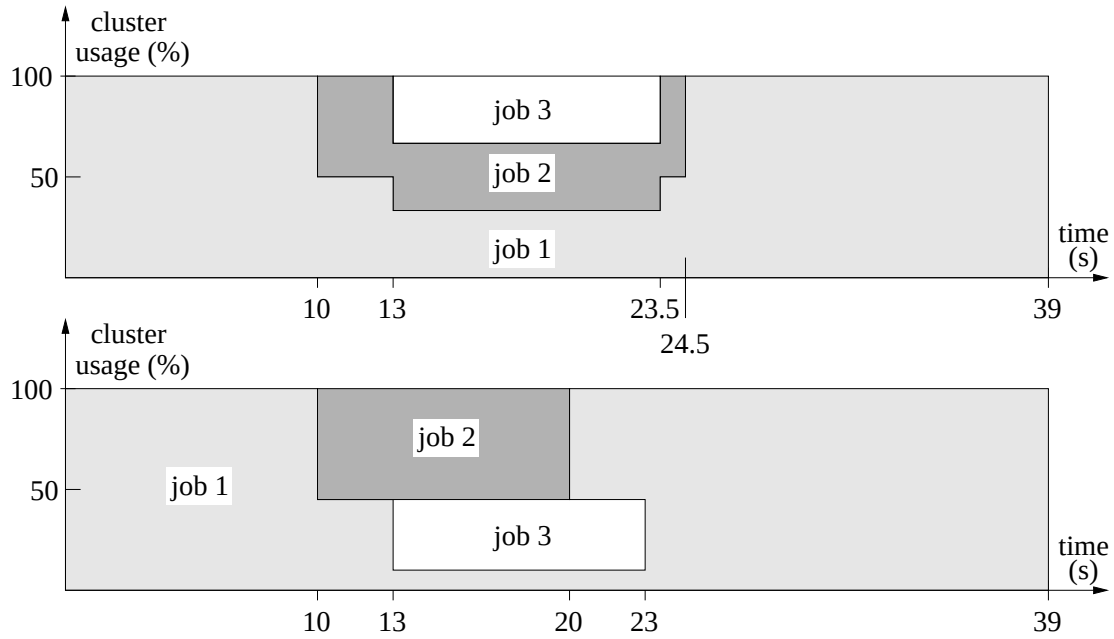


Figure 2.2: Comparison between PS (top) and HFSP (bottom), with jobs that do not require the full cluster.

Figure 2.2 compares the PS (top) and HFSP (bottom) scheduling disciplines. With HFSP, job j_2 would preempt job j_1; since j_2 requires only 55% of the server, the remaining 45% can still be used by j_1. When job j_3 arrives, it would preempt job j_1 (but not job j_2), but it is sufficient to allocate 35% of the system to serve it, leaving 10% of the server to job j_1. As shown in the Figure, the mean job completion time under HFSP is smaller than that achieved by PS, but system resources are allocated such that no job is "mistreated." Note that the final order of job completion with HFSP is different from that achieved by PS (j_2, j_3 and j_1 instead of j_3, j_2 and j_1): in this case job j_2 finishes before the corresponding completion time in case of PS, therefore the fair allocation of the resources is not compromised.

2.2.2 Hadoop MapReduce

MapReduce, popularized by Google with their work in [18] and by Hadoop [2], is both a programming model and an execution framework. In MapReduce, a job consists in three phases and accepts as input a dataset, appropriately partitioned and stored in a distributed file system (namely, HDFS). In the first phase, called Map, a user-defined function is applied in parallel to input partitions to produce intermediate data stored on the local file system of each machine of the cluster; intermediate data is sorted and partitioned when written to disk. Next, during the Shuffle phase, intermediate data is "routed" to the machines responsible for executing the last phase, called Reduce. In this phase, intermediate data from multiple mappers is sorted and aggregated to produce output data which is written back on the distributed file system.

In Hadoop MapReduce, the JobTracker takes care of coordinating TaskTracker nodes, which can be thought of as the worker machines. A key component of the JobTracker is the scheduler, which is the subject of this work. The role of the scheduler in MapReduce is to allocate TaskTracker resources to running tasks: Map and Reduce tasks are granted independent *slots* on each machine. The number of Map and Reduce slots on each TaskTracker is a configurable parameter, which depends on the cluster in which Hadoop is deployed, and on the characteristics (e.g., the number of CPU cores) of each server in the cluster.

When a *single* job is submitted to the cluster, the scheduler simply assigns as many Map tasks as the number of available slots in the cluster. Note that the total number of Map tasks is equal to the number of partitions of the input data. It is important to notice that the scheduler tries to assign Map tasks to slots available on machines in which the underlying storage layer holds the input intended to be processed, a concept called *data locality*. Also, the scheduler may need to wait for a portion of Map tasks to finish before scheduling subsequent mappers, that is, the Mapphase may execute in multiple "waves", especially when processing very large data. Similarly, Reduce tasks are scheduled once intermediate data, output from mappers, is available.[1]. When *multiple* jobs are submitted to the cluster, the scheduler decides how to allocate available task slots across jobs.

[1]Precisely, a configuration parameter α indicates the fraction of mappers that are required to finish before reducers are awarded an execution slot.

The default scheduler in Hadoop implements a FIFO policy: the whole cluster is dedicated to individual jobs in sequence; optionally, it is possible to define priorities associated to jobs. In practice, the FIFO scheduler works as follows: it assigns tasks (Map or Reduce) in response to heartbeats sent by each individualTaskTracker, which report the number of free Map and Reduce slots available for new tasks. Task assignment is accomplished by scanning through all jobs that are waiting to be scheduled, in order of priority and job submission time. The goal is to find a job with a pending task of the required type (Map or Reduce). In particular, for Map tasks, once the scheduler choses a job, it will select greedily the more suitable task to achieve *data locality*.

Another scheduler implemented in Hadoop is the Hadoop Fair Scheduler, which we call FAIR [2]. FAIR was developed at Facebook, and that is used in many production environments. It groups jobs into "pools" and assigns each pool a guaranteed minimum share of cluster resources, which are split up among the jobs in each pool. In case of excess capacity (because the cluster is over dimensioned with respect to its workload, or because the workload is lightweight), FAIR splits it evenly between jobs. The scheduling algorithm implemented in FAIR works as follows: it divide each pool's minimum share of resources among its jobs, and it divides eventual excess capacity among *all* jobs. When a slot on a machine is free and needs to be assigned a task, FAIR proceeds as follows: if there is any job below its minimum share, it schedules a task of that particular job. Otherwise, FAIR schedules a task that belongs to the job that has received less resource, based on the notion of "deficit."

Finally, the Capacity Scheduler from Yahoo offers similar functionality to the Fair Scheduler but takes a somewhat different philosophy. In the Capacity Scheduler, you define a number of named queues. Each queue has a configurable number of map and reduce slots. The scheduler gives each queue its capacity when it contains jobs, and shares any unused capacity between the queues. However, within each queue, FIFO scheduling with priorities is used, except for one aspect – you can place a limit on percent of running tasks per user, so that users share a cluster equally. In other words, the capacity scheduler tries to simulate a separate FIFO/priority cluster for each user and each organization, rather than performing fair sharing between all jobs. The Capacity Scheduler also supports configuring a wait time on each queue after which it is allowed to preempt other queues' tasks if it is below its fair share.

2.3 Hadoop Fair Sojourn Protocol: Design

In this Section we present the design of a size-based scheduling protocol inspired by the FSP discipline, and its integration in Hadoop.

In presenting our work, we take a top-down approach: first we give a general overview of the scheduler architecture, then we explain in detail the most relevant components of our scheme.

[2]Its official acronym is HFS, but since our scheduler has a similar acronym, we will use FAIR instead; `http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html`

2.3.1 General Architecture

The design and implementation of a new scheduling component for Hadoop is a delicate task, as scheduling decisions are responsible for the performance achieved by MapReduce jobs. Furthermore, it is important to recall that scheduling decisions are made by the JobTracker, which runs in a single node: as such, the scheduler design (both the architecture and the underlying resource allocation algorithm) need to be simple to avoid creating a bottleneck in the system.

Figure 2.3, illustrates a the HFSP scheduler architecture. In order to explain the role of each component, in the following we highlight the key problems that need to be addressed in the implementation of HFSP.

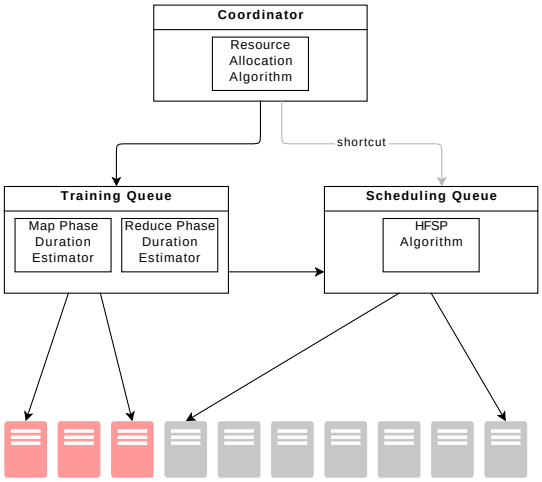


Figure 2.3: Sketch of the HFSP Architecture.

Job size estimation. HFSP belongs to the family of size-based schedulers, in which job size is *assumed to be known in advance*. Clearly, this does not apply in the context of MapReduce: as a consequence, HFSP requires a component that is used to estimate job sizes. The goal of HFSP is to avoid at all costs to "waste" work and cluster resources to perform job size estimation: for this reason we develop a mechanism to compute approximate task length *while they run in the cluster*, thus contributing to the job execution process. This is performed by a component called **Training Queue**.

Another problem we address is resource allocation, which amounts to deciding – in a dynamic manner – how to partition cluster machines between those dedicated to job size estimation and those assigned to the HFSP scheduler, in which jobs are eventually served and complete their execution. To do so, we develop a dedicated component, labelled **Coordinator**, which we describe in Sect. 2.3.2. We dedicate Sect. 2.3.3 to describe the job size estimation mechanism.

Job Preemption. The HFSP scheduling discipline, similarly to the original FSP scheme, is based on the concept of preemption: a new arriving job can preempt a running job, which is then resumed when resources become available. A preemption primitive is not available in Hadoop: instead, the commonly used technique to

approximate job preemption is that of "killing" tasks or entire jobs. Clearly, this is not optimal, because a potentially large fraction of work (including CPU utilization and, most importantly, I/O utilization) is wasted. To overcome this limitation, we have developed two new primitives, namely Suspend and Resume, which handle in an efficient way preemption and the eventual materialization on disk of the state of suspended tasks and jobs.

We present the design of the Suspend and Resume primitives in Sect. 2.3.4.

Scheduling Algorithm. The original FSP discipline is designed for a single-server system, in which jobs have a simple structure. Extending FSP to function in the context of MapReduce is not trivial for the following reasons: *i)* The single serving queue model is not appropriate; instead the scheduling algorithm needs to account for the discrete nature of slots available to execute parallel jobs, which affects how job aging – that tracks how much work has been done for each job in the system – is computed; *ii)* A MapReduce job has a complex structure, composed by Map and Reduce phases, which require their own separate scheduling protocol; *iii)* data locality – that is, making sure that Map and[3] Reduce tasks operate on local data – require special care in taking scheduling decisions.

The component labelled **Scheduling Queue** takes care of scheduling decisions, and we dedicate Sect. 2.3.5 to a detailed overview of the HFSP scheduling algorithm.

Summary: What happens When a Job is Submitted.

We summarize how the HFSP scheduler works, from a high-level perspective. Jobs in MapReduce are composed by two "sub-jobs", corresponding to the Map and the Reduce phases.[4] While the estimation of the duration of the Map phase can be simply done running a subset of the Map tasks (see Sect. 2.3.3), the estimation of the duration of the Reduce phase can not be done until *all* the tasks of the corresponding Map phase have completed. For this reason, the Map and the Reduce phases are treated as if they were two separated jobs.

It is possible to identify four different steps that has to be done to complete a job:

- When a job is submitted to Hadoop, it is sent in the Training Queue in order to estimate the duration of the Map phase;
- When the estimation is complete, the information is given to the HFSP scheduler, which decides when the other Map tasks will be executed;
- When a fraction α of all the Map taks have completed, Hadoop needs to create the Reduce tasks; the allocation request is sent to the Training Queue in order to estimate the duration of the Reduce phase; if α is close to one (e.g., $\alpha = 0.95$), then the probability that the Map would be preempted is very low, therefore any Reduce task can retrieve the necessary data for all the Map tasks;

[3]In general, data locality does not apply to Reduce tasks. As it will be clear in the following, this is not true for HFSP.

[4]We associate the Shuffle phase to the Reduce sub-job.

- When the estimation is complete, the information is given to the HFSP scheduler, which decides when the other Reduce tasks will be executed;

Setting the value of α close to one may synchronize the Shuffle phase, creating possible bottlenecks in the network and slowing down the transfers. Recent studies [] shows that the network does not actually represent a bottleneck, therefore the synchronization does not hurt the overall performance. On the other hand, this approach offers a clear advantage: when the Reduce tasks are executed, all the Map tasks have already completed, therefore the Reduce tasks start immediately to work, instead of occupying task slots waiting for the data, as it happens with the basic FIFO and FAIR schedulers.

Clearly, the time required to complete size estimation plays a crucial role: the faster an estimate is established, the quicker the jobs can be effectively scheduled to receive an appropriate amount of cluster resources.

The speed of the estimation in the Training Queue depends on the machines assigned to such a queue, which are assigned by the Coordinator dynamically such that to guarantee, under full load, full cluster utilization.

2.3.2 The Coordinator

The role of the Coordinator is that of assigning task slots – which are the basic execution unit in Hadoop – to the training and scheduling queues respectively. Note that the Coordinator is not responsible for deciding the size of the sample for the job length estimation, which is done as described in Sect. 2.3.3. Additionally, recall that task slots and hence both training and scheduling queues, must be handled separately (and differently) between Map and Reduce tasks.

A simple approach to resource allocation would be to dedicate a fixed fraction of servers – that we label *training share* – to the training queue, no matter what is the cluster occupation. However, a static assignment of resources is sub-optimal, especially in the realistic case in which the whole system is in a *transient phase*: a simple clarifying example explains why. Assume no jobs are in execution and that the scheduling queue is empty: when a new job arrives, the majority of cluster resources – those dedicated to jobs whose size is known and that can be scheduled – remain idle. This situation is exacerbated by a bursty arrival of new jobs, and its severity depends on the training share. Furthermore, assume that there are no new jobs arriving in the system: if the training share is too large, all the corresponding task slots on those machines would be wasted, as they cannot be used by the scheduling queue.

In our implementation of the Controller, the partitioning and allocation of cluster resources to training and scheduling queues is only logical. In particular, available slots are "tagged" with a label indicating whether they are executing a task as part of the training or the scheduling queues. With a logical partitioning of task slots it is simple to provide a dynamic approach to resource allocation.

We are now ready to explain how the Controller operates under different scenarios.

The training queue claims task slots. This case arises for two reasons. (i) The cluster is under-utilized because there are no jobs in the scheduling queue; as such, the Controller allocates more resources to the training queue, to speed-up job size estimation.[5] (ii) The scheduling queue occupies more resources than it should, e.g. because no new jobs arrived and training resources are diverted to the scheduling queue; as such, the Controller allocates task slots to the training queue, to re-establish cluster partitioning according to the training share.

Clearly, if the scheduling queue uses exactly the training share, and the training queue is full, all new jobs that arrive are queued until training slots free up, or the Coordinator detects an opportunity for the training queue to claim task slots from the scheduling queue.

The scheduling queue claims task slots. This case arises when task slots are assigned to the training queue, due to underutilization of the scheduling queue. As such, the Controller allocates task slots to the scheduling queue, to re-establish cluster partitioning according to the training share parameter.

In assigning the tasks to the Training Queue or the Scheduling Queue, the Coordinator needs to deal with the re-assignments, *i.e.*, how to switch a running task from one queue to another. The Coordinator has the following choices: either Kill a running task, or Wait for a running task to complete. Clearly, Kill operations are utterly expensive as all work done by a task is lost. Furthermore, note that Kill operations are especially costly for Reduce tasks. As such, the Coordinator operates on running tasks solely using Wait (a choice adopted in other schedulers [37]) for both Map and Reduce tasks. The waiting time is limited, because Map tasks are generally very short, and Reduce tasks in the training phases are executed for a pre-defined amount of time (as explained in more detail in Section 2.3.3).

There is a notable exception to the behavior of the Controller described so far. In case of extremely small jobs, *i.e.*, jobs composed by less than five Map tasks, the sample size for the training phase would be equal to the job itself. For this reason, extremely small jobs are sent directly to the Scheduling Queue (see the shortcut path in Fig. 2.3).

2.3.3 The Training Queue

The HFSP scheduler, as explained in Sect. 2.3.1, treats the Map and Reduce phases separately. Let $\mathcal{M}_i$ and $\mathcal{R}_i$ the set of tasks associated to the Map and Reduce phases of job i respectively. We indicate with $\sigma(\mathcal{M}_i)$ the total duration, in units of time, of the Map phase and with $\sigma(\updownarrow_i)$ the duration of a single Map task (similar definitions apply in the case of Reduce phase).

Estimating the size of the Map phase. We posit that the Map phase duration $\sigma(\mathcal{M}_i)$ is the sum of the duration of all Map tasks.

We observe[6] that, across a variety of jobs, Map tasks are generally small, *i.e.,* they take a relatively stable, short time to execute. Now, how many "samples"

[5]Recall that all tasks executed in the training phase contribute to the job process: the fraction of work done for a job when estimating its size is not wasted.

[6]The work in [37, 12] confirm this observation.

Map tasks of a job should be scheduled for execution in the training queue, for computing an estimate of the whole duration of the Map phase? The number of samples to be used is a trade-off between the estimation speed and accuracy. The following expressions indicate, respectively, the "true" duration $\sigma(\mathcal{M}_i)$ of the Map phase, and its estimate $\theta(\mathcal{M}_i)$:

$$\sigma(\mathcal{M}_i) = \sum_{j \in \mathcal{M}_i \setminus \mathcal{T}_i} \sigma(m_j) \tag{2.1}$$

and

$$\theta(\mathcal{M}_i) = \big(|\mathcal{M}_i| - |\mathcal{T}_i|\big) \frac{\sum_{k \in \mathcal{T}_i} \sigma(m_k)}{|\mathcal{T}_i|} \tag{2.2}$$

where $\mathcal{T}_i$ is the sample set, that is the set of Map tasks scheduled for execution in the training queue, and $|\cdot|$ indicates the set cardinality. Note that for both the "true" and the estimate duration of the Map phase, we deduce the amount of work done in the training queue.

We have empirically observed that, using different data center traces, a sample set equal to five Map tasks provide sufficiently high accuracy (cf. Sect. 2.4.3 for details)

Estimating the size of the Reduce phase. Similarly to what described for the Map phase, we posit that the duration $\sigma(\mathcal{R}_i)$ is the sum of the duration of all Reduce tasks.

Estimating the duration of the Reduce phase requires a careful approach: the execution time of a Reduce task can be broken down into (i) Shuffle time – that is, the time it takes to move output data from mappers to reducers –, (ii) sort time – because in Hadoop, input data to Reduce tasks is always sorted –, and (iii) the time it takes to perform the actual work specified by the Reduce function.

Figure 2.4 illustrates the three stages that contribute to the Reduce task size, where on the x-axis we have time, and on the y-axis we have a measure of progress, in percentage, as measured by some internal components of Hadoop.

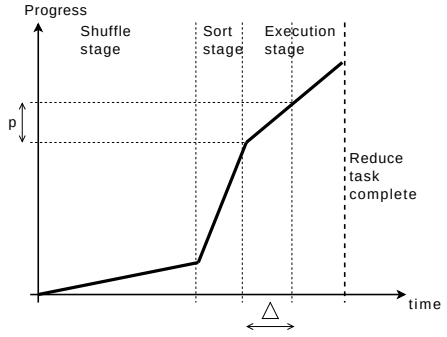


Figure 2.4: Illustration of the three stages that contribute to a Reduce task size (in units of time).

Since a Reduce tasks can be orders of magnitude longer than Map tasks, we aim at providing an estimation of the duration before the tasks in the training set

actually complete. Let $\tilde{\sigma}(r_k)$ estimate of the execution time of a Reduce task r_k; as a first approximation, we ignore the Shuffle and sort times, and we compute $\tilde{\sigma}(r_k)$ as follows:

$$\tilde{\sigma}(r_k) = \frac{\Delta}{p_k} \; \forall k \in \mathcal{T}_i$$

where Δ is a configurable parameter that sets the trade-off between estimation accuracy and speed, and p_k is the progress done by task r_k during the execution stage. For example, $p_k = 0.1$ indicates that task r_k made a progress of 10% towards its completion. Note that Δ establishes the maximum amount of time a Reduce task will remain in execution in the training queue, which constitutes a bound on the training time.

The following expressions indicate, respectively, the "true" duration $\sigma(\mathcal{R}_i)$ of the Reduce phase and its estimate $\theta(\mathcal{R}_i)$:

$$\sigma(\mathcal{R}_i) = \sum_{j \in \mathcal{R}_i \setminus \mathcal{T}_i} \sigma(r_j)$$

$$\theta(\mathcal{R}_i) = \big(|\mathcal{R}_i| - |\mathcal{T}_i|\big) \frac{\sum_{k \in \mathcal{T}_i} \tilde{\sigma}(r_k)}{|\mathcal{T}_i|}$$

where r indicates an individual Map task, $\mathcal{T}_i$ is the sample set, that is the set of Reduce tasks scheduled for execution in the training queue, and $|\cdot|$ indicates the set cardinality.

In Section 2.4.3, we discuss the accuracy of our estimator using data center traces.

2.3.4 Preemption

With the HFSP discipline, jobs are served in series: therefore, is a long job is running and a new small job arrives, the long job should be preempted: cluster resources are "released" by running jobs, and acquired again when the new jobs completes.

Preemption can be implemented in different ways. For instance, a running task can be killed: as previously discussed (cf. Sect. 2.3.2) Kill operations are expensive as all work done by a task is lost. Another alternative is to Wait for a running task to complete, as done in [37]. If the running time of the task is small (an information available from the training phase), then then the waiting time is limited, and this choice does waste resources or the work done by the running tasks.

While the Wait method is easy to implement and provide good results, there are cases – tasks with long running time – where the delay introduced by this approach may be too high. In these cases, we adopt an approach that we have called **eager preemption**: such an approach requires new preemption primitives, namely Suspend and Resume, that are not available in Hadoop. In order to implement these primitives in Hadoop, the key principle we use is to delegate to the operating system (OS) everything that is related to *context switching*. The HFSP scheduler operates on the *child java virtual machine* (JVM) that is fired by the parent JVM – namely the TaskTracker – to execute a particular Map or Reduce task. The child JVM is

effectively a process, which can be suspended and resumed using standard POSIX signals, namely `SIGSTOP` and `SIGCONT`. The eager preemption module of HFSP does not handle context (precisely, process) switching: it is the operating system that is in charge of moving the context of a suspended process in RAM, and eventually proceeding with its materialization on disk (in the swap partition).

We note that our implementation requires to introduce a new set of states associated to an Hadoop task, the relative messages for the JobTracker and TaskTracker to communicate eventual state changes and their synchronization. Furthermore, it is important to discuss two important details:

- *Impact on data locality*: generally, data locality only affects Map tasks. Instead, with eager preemption, the HFSP scheduler also takes care of data locality for Reduce tasks: indeed, when a job and its tasks need to be resumed, it is important to do so on the *same machines* in which they were suspended.

- *Side effects*: eager preemption should be used with care in case of MapReduce jobs that operate on "external" resources, e.g. that heavily use Hadoop *streaming* or *pipes*. Our implementation can be easily extended to provide API support to inhibit Suspend and Resume primitives for such particular workloads.

2.3.5 The HFSP Scheduling Algorithm

In this Section we describe our HFSP scheduling algorithm, discussing the main issues that arise in extending the basic FSP algorithm to a multi-server (that is, parallel) setting. Note that we gloss over the formalism adopted in the original work on FSP [19]: concepts related to *dominance* of a discipline with respect to another, and the related formal proofs that apply to HFSP are deferred to an extended version of this work. Instead, here we focus on systems aspects of the HFSP scheduler.

The issues introduced by a multi-server setting can be summarized as follows. First, resource allocation needs special care to avoid cluster under-utilization. Second, the function to compute job aging – that is to track the work progress of each job – is more complex due to the nature of MapReduce jobs. Finally, event handling in Hadoop also require special care, especially to synchronize job state to avoid inconsistencies.

Algorithm 1 provides an high-level description of the HFSP algorithm. We use the term *job* to indicate a Map or a Reduce sub-job, since the two phases are treated separately as explained in Sect. 2.3.1.

Essentially, the algorithm is divided in two parts. The first part executes every time a new job arrives or leaves (because it completes or fails) the system; the HFSP algorithm "simulates" what would happen if the scheduler was to behave as a processor sharing discipline, computing an appropriate resource allocation and keeping track of the amount of work done by each job. Then the algorithm sorts jobs according to their projected finish time in the simulated system, which is used to take scheduling decisions in the "real" cluster.

Algorithm 1 The HFSP algorithm

```
1. while a job is submitted / finishes do
2.    for all jobs do
3.       compute the max-min fair share
4.       apply job aging function
5.    end for
6.    sort jobs according to their finish time in virtual time
7. end while
8.
9. Request resources to the Coordinator
10. Wait or Eager preempt running jobs
11. while a task slot is available on machine M do
12.    for j in jobs do
13.       if j has an unlaunched task t then
14.          launch t on M
15.          return
16.       end if
17.    end for
18. end while
```

The second part executes when the scheduler claims resources, by requesting them to the Coordinator, or when a free task slot is available and gets assigned to the tasks of an active job, as selected by the scheduler.

We now describe, in details, the internals of the HFSP algorithm. Similarly to what for many variants of fair queueing disciplines, we introduce the concept of *virtual time* and, specific to our setting, *virtual cluster*. The virtual cluster is used to simulate a processor sharing scheduling discipline: as such, both resource allocation and a job "aging" function needs to be defined.

Resource allocation. Virtual cluster resources need to be allocated following the principle of a fair queuing discipline. Since jobs may require less than their fair share, in HFSP, resource allocation in the virtual cluster uses a *max-min fairness* discipline. Max-min fairness is achieved using a round-robin mechanism that starts allocating virtual cluster resources to small jobs. As such, small jobs are implicitly given precendence in the simulated cluster, which reinforces the idea of scheduling small jobs as soon as possible.

Job aging. The HFSP algorithm keeps track, in the virtual cluster, the work done by each job in the system. Initially, the size (expressed in time units) of each job is that computed by the training module explained in Section 2.3.3.[7] Each job arrival or departure triggers a call to the job aging function: in HFSP we keep track of the time difference between such events, consider it as a total amount of work done by all jobs in the virtual cluster, and subtract this amount to each task of each job, as decided by the max-min fairness allocation rule.

Recall that jobs are sorted according to the remaining amount of work to be done: as such, it is important to stress that job arrivals do not modify the order computed

[7]Recall that the initial estimate accounts for the work done during the training phase.

for existing jobs in the system. Hence, the strain on the scheduler computational resources is minimal. Furthermore, in our current implementation of HFSP, the job size estimation is only done during in the training queue: we do not use task execution times (in the scheduling queue) to improve estimation accuracy. This is done to avoid unfairness among jobs in the job size estimation.

We conclude this Section by remarking that the HFSP algorithm is applied, separately, to both the Map and the Reduce phase. The main difference between such phases lies in the how job size estimation is done.

2.4 Experiments

This Section is dedicated to a comparative analysis of several scheduling disciplines, including the default FIFO scheduler, FAIR and HFSP. Currently, we implemented HFSP for Hadoop 0.20.205, which is the stable release of Hadoop, used in production environments[8]. We use this stable release to compare the performance of each scheduler.

Next, we specify the experimental setup for our comparative analysis – which include our cluster configuration, and, more importantly, the workloads we used in our experiments, as discussed in Sect. 2.4.1 – and present a series of results in Sect. 2.4.2. In Sect. 2.4.3, we provide additional remarks to understand the how the inner mechanisms of HFSP work.

2.4.1 Experimental Setup

In this work we use both a small local Hadoop test-bed, and a larger deployment using Amazon EC2[1]. For both clusters, the HDFS block size is set to 128 MB; otherwise, they can be described as follows:

- **Small Local Cluster**: 20 machines with 6 cores and a single 1 TB disk per node, 1 Gbps Ethernet. The main Hadoop configuration parameters are as follows: we set 4 Map slots and 2 Reduce slots per node.
- **Large Amazon Cluster**: 100 "`m1.xlarge`" EC2 instances with the following features: each node has four 2 GHz cores (eight virtual cores), 4 disks that provide roughly 1.6 TB of space, and 15 GB of RAM.[9] The main Hadoop configuration parameters are as follows: we set 4 Map slots and 2 Reduce slots per node.

Workloads. Generating realistic workloads to analyze the performance of scheduling protocols is a difficult task, that has only recently received some attention [12, 10, 11]. In this work, we build upon previous efforts that define a thorough methodology to obtain informative workload suites. We use a patched[10] version

[8] We are aware of current efforts to support preemption in future versions of MapReduce [3]: in our future work we will study how HFSP can be ported to such new systems.

[9] Which is the same configuration used in [37].

[10] We provide details on our branch of SWIM in `http://goo.gl/OKWXg`.

Table 2.1: Distribution of job sizes in our workload, as derived from the Facebook dataset.

Bin	**Map tasks**	**Reduce tasks**	**Number of jobs in benchmark**
1	1	-	38
2	2	-	16
3	10-20	2-3	14
4	30-70	-	8
5	75-125	-	6
6	150-250	30-60	6
7	300-500	-	4
8	700-900	75-150	4
9	1500-2000	200	2
10	3000-4000	-	2

of SWIM [12], that comprises workload generation and data generation tools. In our work, a workload expresses in a concise manner *i)* job inter-arrival times, *ii)* a number of Map and Reduce tasks per job, and *iii)* job characteristics, including the ratio between output and input data for Map tasks. Table 2.1 summarizes the main traits of the workloads we used in our experiments. Our patched version of SWIM uses the same input dataset (obtained from Facebook, as described in [12]) used in previous works. The job inter-arrival times is a random variable with an exponential distribution, and a mean of 14 seconds, making the total submission schedule 24 minutes long. Note that, with respect to the work in [37], our workloads are I/O intensive only: we do not report here results for CPU intensive, nor mixed workloads; we will include them in an extended version of this work.

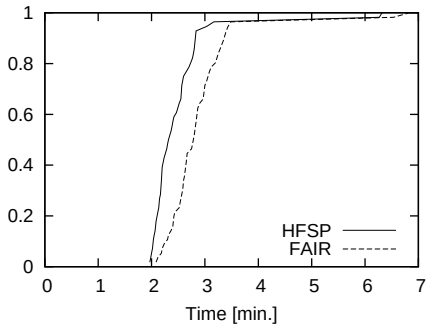
Furthermore, the input data generation we use in our experiments is different from that of the original SWIM implementation. For each job in our workload, we generate an *individual* input file and store it in HDFS, as opposed to using a single input file dimensioned for the largest job in the workload, and then select random HDFS blocks of the same file to create input for the other jobs.

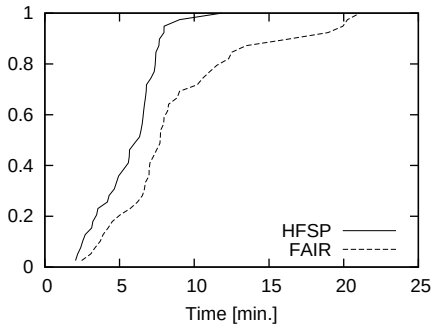
HFSP configuration. The scheduler presented in this work requires tuning a small number of parameters, that we summarize in Table 2.2. The number of Map and Reduce tasks for training is the size of the sample drawn from each submitted job. If a job is smaller than "Shortcut threshold," it is directly sent to the scheduling queue. The number of slots in the cluster dedicated to the training queue is dynamic, and its maximum value is equal to the parameter "Training Share."

Parameter tuning has been done using a manual process. Given the clusters configuration and the workload described above – and based on a series of preliminary experiments to validate our choices – the parameters in Tab. 2.2, are those

Table 2.2: Main configuration parameters used for our experiments, used for the Amazon and Local cluster configurations.

Parameter name	**Amazon**	**Local**
# Map tasks for training	5	5
# Reduce tasks for training	5	5
Reduce time for training	60 sec.	60 sec.
Training Share Map tasks	60 slots	20 slots
Training Share Reduce tasks	30 slots	10 slots
Shortcut threshold (# Map)	10	5
Shortcut threshold (# Reduce)	5	5





that provided the best results. We believe parameter tuning can be automated in a simple way, following the discussions we provide in Sect. 2.4.3. As part of our future research agenda, we will explore the possibility to define an analytical model of HFSP such as to achieve optimal parameter tuning.

2.4.2 Results

We now set off to describe the results we obtained with our experiments. First, we will focus on the workload described in Table 2.1, that we refer to as the FB-dataset. For this series of experiments we used the Amazon Cluster, which is suitable for a

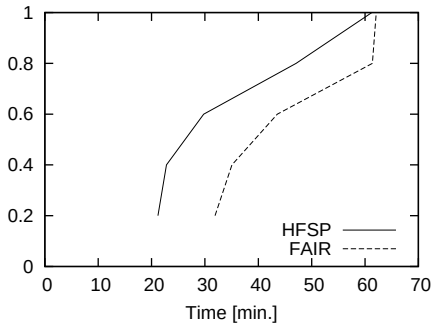


Figure 2.5: ECDFs of sojourn times for the FB-dataset. Jobs are clustered in various bin ranges, as in Tab. 2.1. HFSP improves the sojourn times in most cases. In particular, for small jobs, HFSP and FAIR are roughly equivalent, whereas for larger jobs, sojourn times are 10% to 50% shorter for HFSP with respect to FAIR.

comparative analysis of the HFSP and the FAIR scheduler, since it was also the choice made in [37].

Figure2.5 illustrates the empirical cumulative distribution function (ECDF) of the job sojourn times, which are computed as the time difference between job completion and arrival. We clustered the results according to job sizes, corresponding to the bins in Tab. 2.1. Note also that we omit the ECDF for the FIFO scheduler, for the sake of readability. The most important percentiles for job sojourn times under this discipline are as follows: the $5-th$ percentile is 290 sec., the $25-th$ percentile is 1886 sec., the median is 2402 sec., the $75-th$ percentile is 2607 sec., and finally the $95-th$ percentile is 2843 sec.

Based on the intuition we present in Sect. 2.2, the results we obtain may seem counter intuitive: we would expect short jobs to spend less time in the system when using HFSP instead of FAIR, while long jobs should experience similar sojourn times. Our results, instead, indicate that the improvement of HFSP over FAIR is more evident for medium and long jobs (cf. Figs. 2.4.1 and 2.4.1). The reason for these results lies in the mix of jobs in the FB-dataset, which is biased toward extremely small jobs. In a cluster with 400 Map slots available, the fair share given to extremely small jobs is greater than their requirements in terms of number of tasks, therefore the behavior of the cluster in case of HFSP and FAIR is the same (for this category of jobs). In addition, very small jobs (with 1-2 map tasks) are scheduled as soon as a slot becomes free (both under the HFSP and FAIR scheduling disciplines), and therefore their sojourn time depends almost solely on the frequency at which slot free-up and on the cluster state upon job arrival.

For medium and large jobs, instead, since a single job may occupy the whole cluster, the advantage of HFSP, which schedule the jobs according to their sizes, becomes more clear.

While in Fig. 2.5 we show the gain of HFSP over FAIR for the ensemble of the FB-dataset, we have to show that each individual job performs better with HFSP than the corresponding sojourn with FAIR. If this is the case, then we show that HFSP is able to provide fairness, trying to minimizing the sojourn time. To this

aim, we compute the difference between the sojourn time with FAIR and with HFSP for each individual job. Figure 2.6 shows such differences for the different jobs.

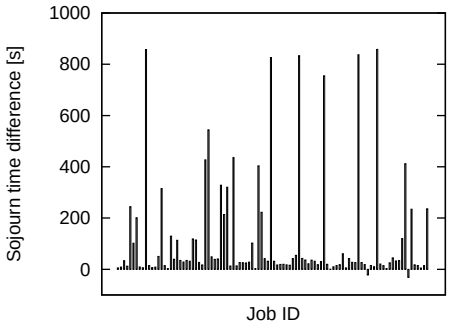


Figure 2.6: Difference between the sojourn time with FAIR and with HFSP for each individual job.

We note that in our experiments, there were two jobs (with a single Map task), that exhibit a slightly better sojourn time (roughly 20 sec., with respect to a job duration of 180 seconds) in FAIR than in HFSP. This result seem to violate the *principle of dominance* stated in [19]: we stress that HFSP is a practical implementation of FSP in a multi-server setting, whereby we do not assume job size to be known in advance. As such, the introduction of a training phase prior to the actual scheduling, requires the original dominance theorem to be re-formulated; this is a challenging task that falls outside the scope of this article.

Now, as it is possible to evince from the discussion above, the workload obtained using the SWIM tool, using the Facebook dataset is particularly lightweight, in the sense that the vast majority of jobs included in the workload is: *i)* only composed by Map tasks, and *ii)* the number of tasks per job is small. In addition, as noticed in [37], the cluster utilization (especially for the Amazon Cluster) is quite low.

As such, we now present another series of experiments that we carried out on our local cluster (which is smaller, and thus highly utilized) using a different workload. In practice, we use the work presented in [12], whereby a different dataset, this time obtained from a Hadoop deployment at Yahoo!, indicate the presence of several large jobs (in terms of number of Map and Reduce tasks), that is more critical for the three schedulers we examine in this work. Table 2.3 summarizes the workload we used in our experiments, where we restrain our attention to fewer, but larger jobs, that exhibit the same arrival time we discussed in Sect. 2.4.1. Unfortunately, the original Yahoo! trace is not publicly available, hence we used Tab.2 in [12] to synthesize a workload that mimics the characteristics of the real traces.

Fig. 2.7 illustrates the ECDF of the job sojourn times for each job. The workload we describe above is more stressful for the three schedulers we study in this work. In particular, we remark a large number of opportunities (HFSP suspended 119 Reduce tasks) for the preemption primitives we develop as part of HFSP, that belong to two large jobs.

Table 2.3: Synthetic workload (as derived from the Yahoo dataset in [12]). There are 15 jobs, that are sorted based on their arrival time in the system.

Job ID	**Map tasks**	**Reduce tasks**
1	48	45
2	55	22
3	203	12
4	101	60
5	60	60
6	20	12
7	93	1
8	69	45
9	20	2
10	13	11
11	55	17
12	30	26
13	40	1
14	147	32
15	48	13

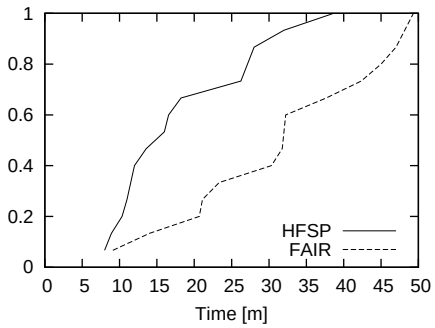


Figure 2.7: ECDFs of sojourn times for the Yahoo-dataset. HFSP strikingly improves with respect to the FAIR scheduler.

2.4.3 Additional Remarks

We now focus on a particular component of HFSP, the Training Queue and study the behavior of the estimators we defined to obtain an approximation of the job duration. This information is used by the HFSP algorithm to perform scheduling decisions and it is important to discuss how the estimation accuracy varies as a function of the number of "samples" that are used to compute job duration.

In the following, we focus on the Map phase, and define the estimation accuracy as follows. For each job i, we compute the "ground truth" mean job duration, as obtained from the logs of our experiments[11], which is defined in Eq. 2.1. Additionally,

[11]Here we focus our attention to the experiments we run on the Amazon Cluster, that is those using the Facebook workload.

we compute the approximate job duration as a function of the number of "samples" that our estimator could take into consideration to build the estimate, using Eq. 2.2, where we let $|\mathcal{T}_i| \in \{1, 50\}\, \forall i$.

Next, we define the estimation error, for each job i, as a function of the sample set size $|\mathcal{T}_i|$:

$$\epsilon_i = \left. \frac{\theta(\mathcal{M}_i) - \sigma(\mathcal{M}_i)}{\sigma(\mathcal{M}_i)} \right|_{|\mathcal{T}_i|}$$

Finally, we compute an aggregate measure of the estimation error we make across all jobs in our workload as follows, $\epsilon = \mathrm{RMSE}\big(\epsilon_i\big)\Big|_{|\mathcal{T}_i|}$, where RMSE stands for the root mean square error, and it is expressed in seconds.

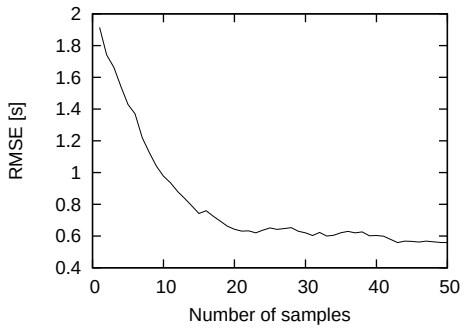


Figure 2.8: Estimation accuracy, expressed as the RMSE across all jobs, as a function of the sample set size.

Fig. 2.8, plots ϵ, as a function of the "sample" set size $|\mathcal{T}_i|$. The figure pinpoints at a diminishing returns effect when the sample set size exceeds the value of 20. Clearly, ϵ tends to zero as the number of samples approaches the number of tasks for each job, but we omit this information for the sake of clarity.

In our experiments, we used a sample set size that is less than 20. As hinted in Sect. 2.3, the selection of a sample set size is a trade-off between accuracy and speed: while an analytical justification for the choice we made is a challenging exercise (that we leave for future work), for our experiments we used an empirical approach that led to the choice we report in Tab. 2.2.

Similar considerations can be made for the selection of another parameter that governs the Training Queue, namely the *training share*. Also in this case, we conjecture that it is possible to come up with an analytical model of the HFSP system to help in selecting an appropriate (and eventually optimal) training share value. However, in this work we proceeded again with an empirical approach in selecting this value.

2.5 Discussion

In this Section, we present several points that complement the work we have presented so far, which we believe important to discuss.

Preemption performance. We now discuss some implications of the preemption primitives (Suspend and Resume) that we implemented for HFSP, since it may be reasonable to argue that they could have an ill effect on job performance and hence on their sojourn time.

When one or more tasks of a job are preempted, the memory that they are using can be claimed other processes executing new tasks scheduled to occupy their slot. In this case, the Operating System (OS) may swap the memory contents to disk. When such preempted task are resumed, the OS reloads in memory the swapped context from disk. As such, the Resume operation may introduce further delays that contribute to a longer job sojourn time. We remark that such delay is bounded: indeed, the memory footprint of a task is limited by the way a MapReduce job is engineered. If a Map task is preempted, the memory it uses is roughly equal to the HDFS block size it operates on (plus any additional data holding temporary information). If a Reduce task is preempted, it is generally the case that the amount of data it uses (and hence memory it occupies) is known a priori. As such, the disk I/O that characterize cluster machines is the main limiting factor that contributes to any additional delays to be added to the sojourn time of a job. Clearly, if the preempted task reside in memory only, then such delay becomes negligible.

Finally, we remark that our implementation of preemption may greatly benefit from "sandboxing" techniques. As part of our future work, we plan to explore sandboxing to bring HFSP closer to be "production-ready".

Job with Different Priorities. The design of HFSP takes as a reference the Processor Sharing (PS) discipline to compute the order of the jobs to be scheduled. In PS, each job receives its equal share of the resources. A natural extension of the work would provide different priorities, or weights, to jobs: in this case, we should consider the Generalized Processor Sharing (GPS), where each job receive an amount of resources in proportion to its weight. For instance, if $\mathcal{J}$ is the set with all the jobs in the system, then job k with weight w_k will receive a fraction $\frac{w_k}{\sum_{i \in \mathcal{J}} w_i}$ of the resources. This computation can be easily incorporated in the job aging computation (cf. Sect. 2.3.5) done by the HFSP algorithm.

Job size estimation. We believe reasonable to be skeptical about the ability of such a simple estimation technique we use as part of HFSP, to correctly estimate job sizes in a broader range of workloads and cluster configurations than those we explored in our experiments. Indeed, task execution time, which contributes to job duration, could be regarded as a highly variable quantity: hence, a simple estimator based on averaging a small sample set may exhibit very low accuracy in general settings.

We remark that in HFSP, the estimator is designed as a pluggable module that could eventually be replaced by more sophisticated estimation techniques, therefore providing more accurate predictions. Furthermore, to the best of our knowledge[12], task execution times are instead fairly stable, and exhibit a variability that is below

[12]Our source of information comes from several discussions we had with engineers from the Amazon Web Services EC2 and EMR teams during Hadoop Summit 2012.

5%, especially for the kind of EC2 instances we used for our experiments with the Facebook dataset.

2.6 Related Work

Since the introduction of the MapReduce framework and its implementation in Hadoop, many works have analyzed and modified the system in order to improve the performance. Such studies consider different aspects of the framework (e.g., resource allocation, impact of the network): in our work, we focus on the job and task scheduling with the aim of optimizing the job completion time without affecting the fairness. Therefore, any approach that tries to optimize different aspects can be considered orthogonal to our solution: for instance, in [17] and [38] the authors optimize the network utilization within a single job, therefore their solution can be integrated seamlessly in our proposed system.

Our solution relies on two basic ingredients: the evaluation of the job size and the job preemption. The inference of the job size has been proposed in [34][35][4], but they do not provide preemption.

In general, the schedulers proposed in the literature provide either fairness or the minimization of some performance index (e.g. delay), without considering these aspects at the same time. For instance, the FAIR scheduler [37] is the de facto standard and provides a processor-sharing like job scheduling. The authors in [33] propose a modification of the fair scheduler, therefore not taking into account the delay. The works [30][24][20][22] focus on allocation of resources, with the aim of fairness, without considering delay optimization.

In [31] the authors study the resource assigment problem (by bidding the available Map and Reduce task slots), without considering the time associated to each task and therefore without optimizing the delay. The authors in [26] design a scheduler that takes into account deadlines, but they assume that the task duration is provided by the user, while in our solution, we infer task duration directly from the jobs. Flex [36] provides a framework for the optimization of any given metric, but it is implemented as modification of FAIR scheduler, therefore it is not clear how it can infer the duration of the job or provide preemption. Theoretical works, such as the one in [9], provide interesting insights, but their over-simplified assumptions makes them hard to apply. In our work, we provide the implementation of the scheduler, with all its components.

2.7 Conclusion

In this work we discuss the design of a Hadoop scheduler that is able to provide fairness, in terms of equal share of the resources, and, at the same time, it tries to minimize the execution time of the jobs. The scheduler needs different components, such as a training queue for estimating the job size, or preemptive primitives, or a task for assigning the available slots to the training queue or to the scheduling queue. We provide the implementation of these components along with the HFSP scheduler.

The experiments performed using a workload generator that takes as input real data center traces show that the HFSP scheduler is able to improve the performance, in terms of execution time, of the submitted jobs, for different and heterogeneous workloads.

Our future work will include the issue of a JIRA and a contribution to the Hadoop community with a “contrib” HFSP module.

3 Prefetching

3.1 Introduction

The main idea of ViPeeR dCDNs is to decrease the load of the origin content server by serving clients from ISP managed caches that have been strategically placed close to the clients. The peers in the peer-assisted dCDNs may be network elements such as network nodes or boxes located at customers premises.

The delivery system consists in a set of content caches delivering content replicas to end-users. Among others, managing the dCDN comes with the management of the caches.

The question here is which video content have to be replicated among the peers so that the downloading process keeps being managed by the dCDNs (and not by the traditional CDN). In order to choose the content to be disposed, a recommender system has been designed.

In this chapter, we are focusing on finding the best algorithms and parameters for recommending content (item). Our study is based on a one year dataset from Orange. We refine the tests given in the D4.3.

3.2 Recommendations

Reformulating the problem, in this part of the project, the goal is to predict items which are VoDs files that users would download in the future.

For this purpose, Orange provides a one year dataset, a log file, containing a list of users linked to the items they chose, with the date of download and the region of the users :

idUser	idItem	Region	Date

The log-based recommendation methods that has been developed to predict items can be grouped in two main categories: popularity-based methods and personalized method. The first ones try to predict items that will be the most downloaded by the whole set of users, whereas personalized methods give unique recommendations for each user.

The log file has been cut into several periods of time (the last week, the two last weeks, the last month, the two last months, etc.) and, for each of them, only users that have downloaded a minimum of 10 videos are kept. Then, in order to test the quality of the different methods, each period has been separated in two parts. A

"training" part which is used to execute the methods and a "test" part to check if predictions are good. It is a chronological split, the "test" part contains the 5 last VoDs downloaded by each user during the period.

In the following sections, the two types of method are tackled. First, we focus on personalized methods. Then, we are interested in the popularity-based methods. Tests and comparisons are done to find the best parameters and best methods.

3.3 Personalized methods

In this part, we address personalized methods that generate personalized prediction to each user. The first considered method is the item-based collaborative filtering method. We first study the variation of its results according to the variation of different parameters. Then, in order to make comparisons, we give the results obtained with other personalized methods including user-based collaborative filtering method and the singular value decomposition based method.

3.3.1 Item-based collaborative filtering

The main idea of the item-based collaborative filtering method is to compare items each other to find items which are close to the ones the current user likes [32].

For the tests, we can vary multiple parameters. First, we will examine the impact of the variation of the period of log on the results. Then, we will vary the recommendation value threshold which allows selecting only the recommendations that have a recommendation value less than the threshold. Finally, we will show the impact of the variation of a fix number of recommendations.

These tests are performed using the following parameters:

- $recThreshold = 1$ (recommended items which recommendation value is less than 1 are not proposed)
- $maxRec = 10$ (maximum number of recommendations by users)
- $minRec = 3$ (minimum number of recommendations by users, even if recommendation_value<recThreshold)

3.3.1.1 Variation of the log period

For these tests, the minimum and maximum number of recommendations per user are respectively 3 and 10. The recommendation value threshold is fixed to 1 (given that recommendation value varies between 0 and 5). All the recommendations that have a recommendation value less than the $recThreshold$ are removed, if the minimum number of recommendation is not reached. To quantify the results given by the different methods for each period, and as previously presented in deliverable D4.3, we measure the Precision (equation 3.1), the Recall (equation 3.2) and the Fscore (equation 3.3). Precision is the rate of good predictions among all the predictions and Recall is the rate of good predictions among all the real future downloads. Fscore is the compromise between the two values.

$$precision = \frac{|predicted \cap real|}{|predicted|} = \frac{TP}{TP + FP} \tag{3.1}$$

$$recall = \frac{|predicted \cap real|}{|real|} = \frac{TP}{TP + FN} \tag{3.2}$$

with TP the number of true positive (correct result), FP the number of false positive (unexpected result) and FN the number of false negative (missing result).

$$Fscore = \frac{2\,(Precision \times Recall)}{Precision + Recall} \tag{3.3}$$

	1 week	2 weeks	1 month	2 months	3 months
Precision	10.3%	9.5%	6.8%	5.7%	5.6%
Recall	7.6%	8.7%	8.7%	7.6%	7.4%
Fscore	0.083	0.086	0.071	0.060	0.059
Mean rec-items number	4.1	5.0	4.4	7.1	7.2
Users number	261	701	1172	1906	2914
Items number in training	1312	3124	5749	10413	15567
	4 months	5 months	6 months	1 year	
Precision	7.9%	6.7%	5.3%	3.8%	
Recall	10%	9.7%	7.9%	5.8%	
Fscore	0.082	0.075	0.060	0.044	
Mean rec-items number	7.2	7.6	8.0	8.3	
Users number	5222	7521	8956	12989	
Items number in training	21629	27644	33205	54212	

Table 3.1: Item-based collaborative filtering method: log period duration.

The Table 3.1 (and Figure 3.1) gives the results of the item-based collaborative filtering method by variating the period of log. The best Precision score corresponds to a period of 1 week, the best Recall corresponds to a period of 4 weeks and the best Fscore corresponds to a period of 2 weeks.

We should ask ourselves what is the value we want to optimize : Recall, Precision or Fscore? With a bad Precision, the risk to prefetch items which will not be downloaded is high. Here, the best precision is only 10.3%, it means that only 10.3% of our caches will be useful. A bad Recall means that there exists content that are not (can not be?) predicted by the method. Here, with a Recall of 8.7%, 91.3% of future downloads will not be served by the dCDN. Only "mean" behavior is really captured, this is a well-know weakness of collaborative filtering methods. The best Fscore catches the parameters with both the best Precision and Recall. As one of the main objective of ViPeeR is to limit the downloads out of the dCDNs, we consider that the prefetching task consists in optimizing the Recall.

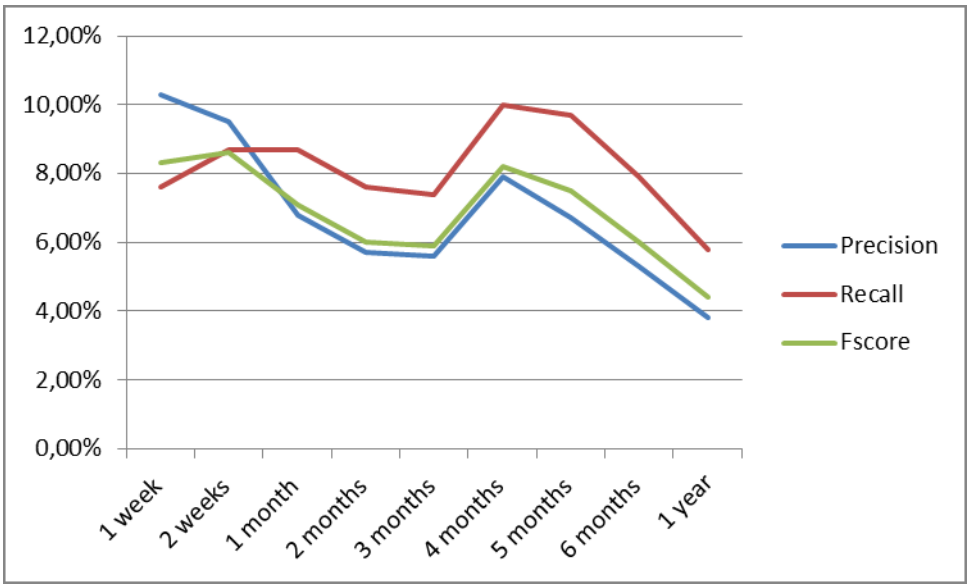


Figure 3.1: Variation of the results of item-based CF according to the log period

3.3.1.2 Variation of the recommendation value threshold

The recommendation value threshold *recThreshold* allows to remove recommended items which the recommendation value is less than this threshold; given that bigger is the recommendation value, higher is the chance for the VoD to be downloaded.

Variation on a 1 week period

	Test 1	Test 2	Test 3	Test4
recThreshold	5—4—3—2	1	0.5	0
Recall	6.5%	7.6%	11.6%	12.7%
Precision	10.9%	10.3%	7.6%	6.4%
Fscore	0.081	0.083	0.088	0.085

Table 3.2: Item-based collaborative filtering method: predicted rate variation, 1 week log period.

The Tables 3.2 and 3.3 show that higher is *recThreshold*, better is the Precision and worse is the Fscore. Moreover, what interest us is that the Recall becomes better as the threshold decreases (it is confirmed on larger period of time). Thus, the recommendation value threshold has no interest since it allows only improving the Precision.

Variation on a 2 weeks period

	Test 1	Test 2	Test 3	Test4
recThreshold	5—4—3—2	1	0.5	0
Recall	6.8%	8.7%	12.6%	14.1%
Precision	10.4%	9.5%	7.7%	7.0%
Fscore	0.081	0.086	0.092	0.094

Table 3.3: Item-based collaborative filtering method: predicted rate variation, 2 weeks log period.

3.3.1.3 Variation of the number of recommendations

The previous part states that *recThreshold* has no interest, since we would like to optimize the recall value. As this threshold was used to vary the number of recommendations by users, it induces that this number will now be the same for each user. In this part, we test multiple values for the number of recommendation that we call *maxRec* on a one week period.

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
maxRec	7	10	20	30	40	50
Recall	10.8%	12.7%	18.1%	21.3%	24.0%	25.6%
Precision	7.7%	6.4%	4.5%	3.6%	3.1%	2.7%
Fscore	0.090	0.085	0.073	0.062	0.054	0.048
	Test 7	Test 8	Test 9	Test 10	Test 11	
maxRec	60	70	80	90	100	
Recall	26.7%	28.4%	29.8%	31.8%	32.7%	
Precision	2.4%	2.2%	2.1%	2.0%	1.9%	
Fscore	0.043	0.040	0.038	0.037	0.035	

Table 3.4: Item-based collaborative filtering method: number of recommendations for each user

According to Table 3.4 (and Figure 3.2), we can see that higher is the number of recommendations, better is the Recall, and worse is the Precision; what seems logical because higher is the number of recommendations, more is the chance to fall on new good predictions but with a big part of bad predictions. Moreover, Fscore becomes worse as the number of recommendations increases.

Actually, the choice of the number of recommendations, whatever the method, will depend on the size of the caches.

3.3.2 User-based collaborative filtering method

In this type of recommendation method, notes on items are computed according to notes of users that are similar to the current user [29]. Then, the items with the

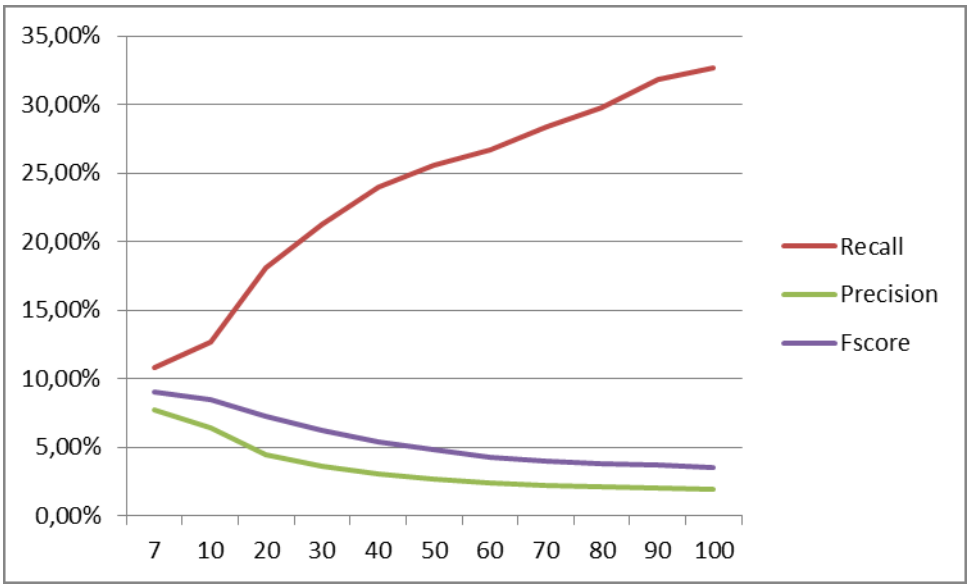


Figure 3.2: Results of item-based CF by variating the number of recommendations

best notes are recommended to the current user.

	1 week	2 weeks	1 month	2 months	3 months
Precision	5.1%	4.1%	3.2%	2.9%	2.2%
Recall	9.6%	7.4%	6.3%	5.8%	4.4%
Fscore	0.066	0.052	0.042	0.039	0.030
	4 months	5 months	6 months	1 year	
Precision	3.7%	3.3%	2.6%	1.8%	
Recall	7.0%	6.4%	5.2%	3.6%	
Fscore	0.048	0.043	0.035	0.024	

Table 3.5: User-based collaborative filtering method.

The Table 3.5 (and Figure 3.3) shows that the user-based collaborative filtering method gives worse results than the item-based one. It confirms what the literature argues generally in comparing these two methods.

3.3.3 Singular Value Decomposition, SVD

A well known method in recommendation is the SVD model-based one. Traditionally, this latter tries to model the users and the items according to implicit features by factorizing the user-item matrix [27].

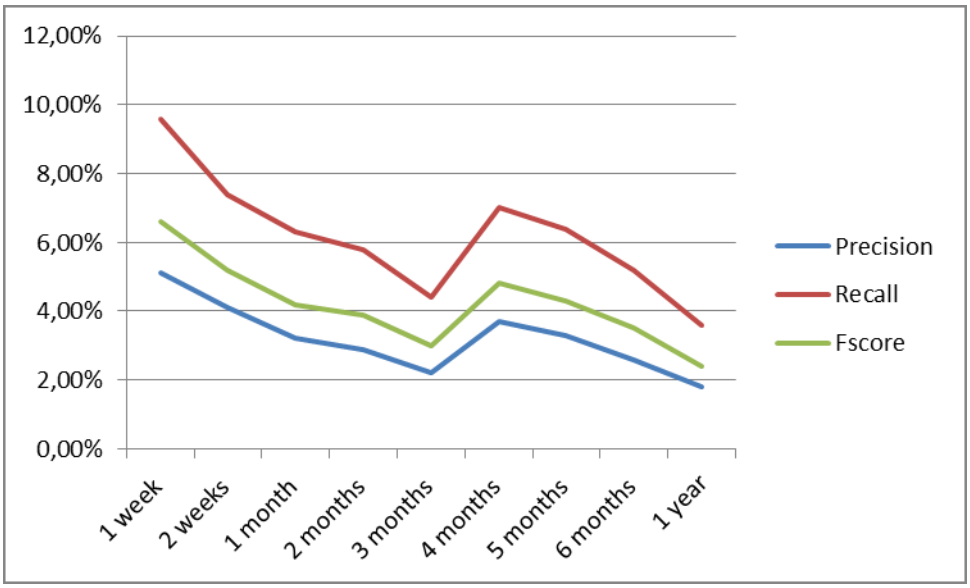


Figure 3.3: Results of user-based CF by variating the log period

3.3.3.1 SVD introduction

Let M be a $m \times n$ matrix linking m users to the n items by interest weights.

$$M = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{pmatrix}$$

SVD is a matrix factorization so that we obtain two matrices U and V containing respectively singular user vectors and singular item vectors, called right and left singular vectors of M. Matrix S is a matrix having singular values of M on its diagonal.

$$M = USV^*$$

where $(V^* = \overline{V}^T)$.

3.3.3.2 SVD interest

After finding U, V and S, a method consists in selecting the k best values of the diagonal of S to obtain a new matrix S_k and the matrices U and V are truncated according to the kept values. Then, the interesting thing is that we can obtain a approximation $\tilde{M}$ of M linking users with items based on a little number of implicit features (the singular values of S_k), so that we will have new interest values on items according to these implicit features.

$$\tilde{M} = U_k S_k V_k^*$$

There are several methods [5][6]to find the different values (using minimization of$\left\|M - \tilde{M}\right\|$, calculating eigenvalues of MM^* and M^*M or others...). We will not detail them here.

3.3.3.3 Results

	1 week	2 weeks	1 month	2 months
Precision	1.4%	0.6%	0.2%	0.1%
Recall	2.9%	1.2%	0.4%	0.2%
Fscore	0.019	0.008	0.002	0.001

Using the tool of Mahout library that allows to realize SVD on the user-item matrix to approximate it, recommendations are much worse than item-based collaborative filtering. This is probably due to the fact we have only 0 and 1 notes on items.

3.4 Popularity-based methods

In the previous part, we have tested different personalized methods for recommendation. The best results for optimizing the recall are obtained with the item-based collaborative filtering method for a period of 2 weeks with a rate of good predictions of about 14% in average. This is quite poor.

In this part, we will focus on more general methods that predict items that will be the most downloaded ones by the whole set of users.

First, two principal popularity-based methods have been realized:

- Simple popularity: a method consisting in only considering the top n most popular items of the log.
- Collaborative Filtering (CF) popularity: a method consisting in considering the top n most popular items among recommendations from the item-based collaborative filtering method.

Then, they have been modified to take into account the region parameter, inducing to new methods: simple popularity by region and collaborative filtering popularity by region. The idea is to regionalize the predictions to see if the region can influence the results. Finally, several attempts to mix the two principal methods have been realized in order to improve the results.

In the following, we first compare the simple popularity and the collaborative-filtering popularity methods on several periods of time. We will also show the variation of the simple popularity by varying the size of top. Then, the results of using simple popularity by region and collaborative filtering popularity by region will be shown. Finally, the methods to mix simple popularity by region and collaborative filtering popularity methods will be detailed.

3.4.1 Simple popularity VS Collaborative Filtering popularity

We compare here the Simple popularity and the collaborative-filtering methods on several periods of time. The top $k = 500$ most popular items are considered in each test. For each period, we measure the Recall value.

The Table 3.6 (and Figure 3.4) shows that the Simple popularity gives better results than the CF popularity, for all the tested period durations. Nevertheless, it

	1 week	2 weeks	1 month	2 months	3 months
CF popularity	33.5%	28.4%	26.7%	20.5%	17.8%
Simple popularity	44.2%	38.8%	36.1%	29.8%	25.4%
Simple_pop ∩ CF_pop	24.9%	24.0%	22.4%	17.6%	15.0%
Items number in training	1312	3124	5749	10413	15567
	4 months	5 months	6 months	1 year	
CF popularity	19.4%	19.9%	16.9%	12.2%	
Simple popularity	25.1%	24.2%	21.8%	18.7%	
Simple_pop ∩ CF_pop	16.0%	15.9%	13.4%	9.8%	
Items number in training	21629	27644	33205	54212	

Table 3.6: Popularity-based methods: Recall variations according to log period duration, top $k = 500$ items considered.

appears that a part of the good predictions from CF popularity are not predicted by simple popularity. Moreover, this table shows that the period of one week gives the best results in terms of good predictions rate for the two methods, with only 1312 items for training.

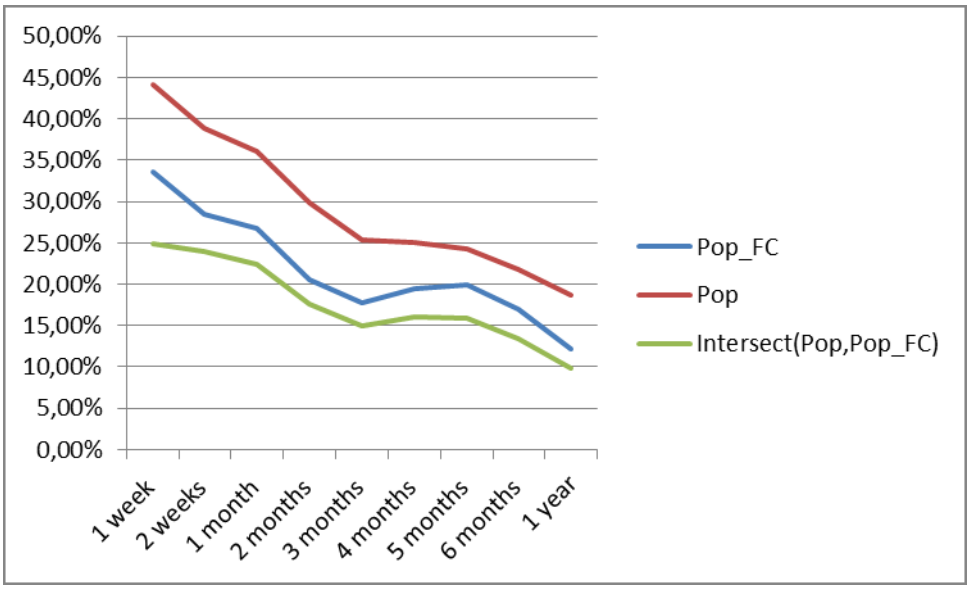


Figure 3.4: Results of simple popularity and CF popularity according to the period of log, top $k = 500$ items considered.

In the following, we would like to see how the Recall varies with the variation of top k, the size most k popular items.

Top	1 week	2 weeks	1 month	2 months	3 months
500	44.2%	38.8%	36.1%	29.8%	25.4%
1000	58.1%	51.9%	48.7%	42.7%	36.6%
2000	65.8%	64.3%	61.3%	55.3%	50.1%
3000	65.8%	74.6%	70.0%	63.7%	57.9%
Items number in training	1312	3124	5749	10413	15567
Top	4 months	5 months	6 months	1 year	
500	25.1%	24.2%	21.8%	18.7%	
1000	35.9%	34.7%	31.7%	26.2%	
2000	49.3%	48.0%	44.5%	36.5%	
3000	57.9%	56.4%	52.7%	44.0%	
Items number in training	21629	27644	33205	54212	

Table 3.7: Popularity-based methods: Recall variations according to k most popular items.

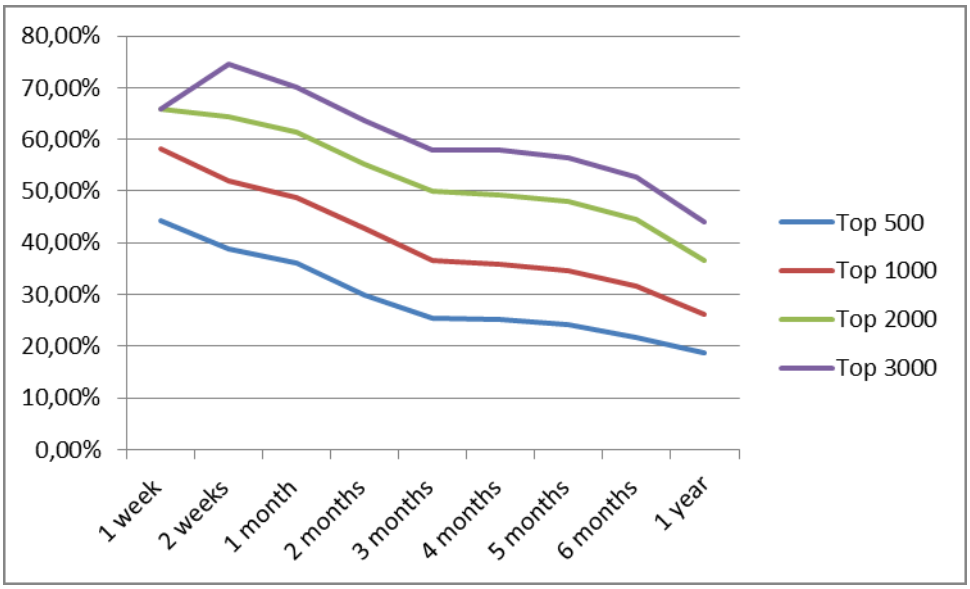


Figure 3.5: Simple popularity with different sizes of top k.

Simple popularity with different sizes of top k

Without looking at this results, it was obvious that bigger is the top k, better is the rate of good predictions because the probability to fall on good predictions increases naturally with the increasing of the number of predictions.

The Table 3.7 (and Figure 3.5) also shows that it is impossible to predict all the future downloads. Indeed, for instance, in the latest week of download (1 week), 34.2% of downloads cannot be predicted because they are unknown in the training dataset, probably due, for the most part, to new videos.

3.4.2 Simple popularity by region

Given that Orange defines 13 regions of users, we are interested in studying the impact of regionalizing the recommendations.

Here, we first test the simple popularity method limited to each region. Users and associated videos have been classified by region. Then, a ranking of the 500 most popular video for each region has been realized.

	1 week	2 weeks	1 month	2 months	3 months
Simple popularity	44.2%	38.8%	36.1%	29.8%	25.4%
Simple popularity by region	23.0%	27.1%	27.2%	24.6%	23.2%
	4 months	5 months	6 months	1 year	
Simple popularity	25.1%	24.2%	21.8%	18.7%	
Simple popularity by region	23.0%	22.8%	20.2%	16.2%	

Table 3.8: Simple Popularity method: Recall taking into account the geographical area or not.

In the Table 3.8, it appears that in average this simple popularity by region method is a bit worse than the normal simple popularity. There is clearly no change in the consumption style between users from different French geographical area. This result is not surprising since movies are proposed for the French people in globality and not for categories of them. This would change certainly if regions were different countries.

3.4.3 CF popularity by region

Here, we would like to see if a collaborative filtering popularity method by region would improve the collaborative filtering popularity method.

	1 week	2 weeks	1 month	2 months	3 months
CF popularity	33.5%	28.4%	26.7%	20.5%	17.8%
CF popularity by region	13.7%	17.4%	18.7%	13.2%	11.6%
	4 months	5 months	6 months	1 year	
CF popularity	19.4%	19.9%	19.6%	12.2%	
CF popularity by region	13.2%	15.2%	13.5%	10.1%	

Table 3.9: CF Popularity method: Recall taking into account the geographical area or not.

In the Table 3.9, we can see again that the regionalization does not give better results. The collaborative filtering popularity method by region is worse than the collaborative filtering popularity method.

3.4.4 Mixing collaborative filtering popularity with simple popularity

In every tests, the simple popularity method is better than the collaborative filtering popularity method. Nevertheless, it appears that each method gives some distinct good predictions (they are in one set but not in the other).

Thus, it would be possible to improve predictions if we can combine the two methods and remove bad predictions from each set.

Let set_1 be the set of the simple popularity predictions and set_2 be the set of the collaborative filtering popularity predictions. Let set_3 be the intersection between the two sets, $set_3 = set_1 \cap set_2$. The following mixing methods have been tested, without exceeding the 500 items recommendation:

1. Method 1: Items from set_3 are kept and the best predictions from set_1 and set_2 which are not in set_3 are equally added.

2. Method 2: Items from set_1 and set_2 are grouped and their popularity value becomes the sum of the popularity values in each set. The items with the best values are selected (top 500).

3. Method 3: We keep only the n% best items from set_1 and the $(100-n)$% best items from set_2.

	1 week	2 weeks	1 month	2 months	3 months
Method 1	43.4%	36.3%	33.4%	27.0%	23.0%
Method 2	41.6%	36.6%	35.3%	28.8%	23.9%
CF popularity	33.5%	28.4%	26.7%	20.5%	17.8%
Simple popularity	44.2%	38.8%	36.1%	29.8%	25.4%

Table 3.10: The two first strategies for mixing Popularity methods: Recall taking into account the log period duration.

In the Table 3.11, we give the best results of method 3 for each period with the optimal value of n.

	1 week	2 weeks	1 month	2 months	3 months
Method 3	47.5%	38.83%	36.4%	-	-
Best n improving the results	60	99	83	none	none
CF popularity	33.5%	28.4%	26.7%	20.5%	17.8%
Simple popularity	44.2%	38.77%	36.1%	29.8%	25.4%

Table 3.11: The third strategy for mixing Popularity methods: Recall taking into account the log period duration.

Method 1 gives better results than collaborative filtering popularity, but is worse than simple popularity.

Method 2 is better than collaborative filtering popularity but is a little worse than simple popularity.

Method 3 gives various results following the value given to n. Most of time, this method is worse than the simple popularity and always better than collaborative filtering popularity. However, sometimes it is possible to find values of n that improve of some points the rate of good predictions. For example, instead of 44% of good predictions with the simple popularity, if $n = 60$ we obtain 47.5% of good predictions on one week of consumption. Nevertheless, for two weeks, this value gives a worse rate of good predictions than using simple popularity method. The choice of the value n is not obvious since it changes every times, even between two identical time periods on two different positions of the log. Indeed, for a period of 1 week taken in the middle of the log file, the best value of n is 89. Moreover, it is not guarantee to find a value of n improving the result as we see in the Table 3.11 for the 2 and 3 months period.

Thus, given that it is not obvious to find the value of n in Method 3 and that Method 1 and Method 2 give bad results in trying to mix collaborative filtering popularity with simple popularity, we will not retain any of them.

3.5 Conclusion

In this part of the project, we aim at predicting the future downloads of each user. We have tested several well-known methods that we can classify in popular methods (CF popularity, simple popularity, CF popularity by region, simple popularity by region and mix popularity) and personalized methods (item-based, user-based, SVD). The popular methods are general and provide k items that should be downloaded by most users. The personalized methods provide personalized recommendations to each user.

The tests show that personalized methods give poor results if they recommend a small number of items. For instance, the best rate of good predictions we obtain by predicting 10 items per user is about 14%, considering a two weeks period of log.

Thus, it seems that it would be preferable to consider more general methods predicting the future downloads of the whole set of users. The popular methods provide a larger number of recommendations and give therefore better results than the personalized ones. Among the popularity methods, the simple popularity is the best since it can predict 44.2% of the future downloads based on a one week period of log.

4 Parallelization of the Genetic Algorithm

4.1 Introduction

In report D4.3, we have introduced the rudiment of the centralized genetic algorithm (GA) for the optimal video chunk placement problem, or the k-product capacitated facility location problem (k-PCFLE). In this chapter, we will detail the parallelization of the GA in MapReduce and its implementation.

4.2 Review of the centralized GA

In the centralized GA, we use the real value encoding to form an individual. The length of each individual is the sum of the storage capacity of all servers. The fitness value is calculated by the objective function in the linear program. Since the placement of video chunks in servers are determined by the individual, the calculation of the fitness value is to find the minimum overall cost to deliver chunks to users. The cost metric in our implementation is the real distance between servers and users. The problem to find the minimum overall cost is then transformed to the Minimum Cost Maximum Flow (MCMF) problem. We use a merging process to realize the crossover operation. The idea is to merge the same section of two parents, where a section represents the chunks stored in one server, so that duplicated gene in two parents has larger probability to stay in the individual. The mutation operation is to select one or more genes in the individual and replace them by other video chunks. The centralized version of GA is described in Algorithm 2. The evaluation() function in the algorithm means the computation of fitness value using MCMF algorithm. The variable t is the number of the current generation, and Q is the set of qualified offspring.

4.3 Parallelizing GA by MapReduce (MR)

The strong incentive of using MR is the huge search space of solutions yielded by k-PCFLP, when we use it to model the real video delivery system with millions of clients and thousands of films. In this section, we first give a brief survey about several models of PGA, then we describe our PGA integrated in MR framework.

Algorithm 2 Genetic Algorithm for k-PCFLP

```
1  t = 0
2  min = ∞
3  Q = ∅
4  Initialize generation G_t
5  for each individual ∈ G_t do
6      if Evaluate(individual) < min then
7          min = Evaluate(individual)
8      end if
9  end for
10 for 1 to N_p do
11     offspring= Mutation(Crossover(Select(G_t)))
12     if Evaluate(offspring) < min then
13         Q ⟵ Q ∪ {offspring}
14     end if
15 end for
16 while Q ≠ ∅ do
17     Replace(G_t,Q)
18     t = t + 1
19     Q = ∅
20     update min
21     for 1 to N_p do
22         offspring= Mutation(Crossover(Select(G_t)))
23         if Evaluate(offspring) < min then
24             Q ⟵ Q ∪ {offspring}
25         end if
26     end for
27 end while
```

4.3.1 Parallel GA (PGA) overview

PGAs are classified on different categories based on parallelizing methods: (i) the way in which the whole population is partitioned, (ii) size of each subpopulation (deme), (iii) exchanging schema of individuals between demes. Traditionally, there are three classes of PGA: master-slave GA, coarse-grained GA and fine-grained GA.

Master-slave PGA uses a single population. The parallelization can be implemented on two operations including the evaluation of fitness function and the mutation, because both of the two operations execute based on the knowledge of a single individual. In this master-slave model, the whole population is stored in the master node, while the slaves evaluate the fitness and apply mutation. Moreover, the fitness evaluation is the most commonly parallelized operation, since it is the most time consuming part of a GA. The parallelization is realized by assigning a fraction of the population to each of the slave node. The communication occurs when slave nodes receive the assigned subpopulation and return the evaluation results to the master node. It is not mandatory for the master node to wait for all the evaluation results to proceed the production of the next generation. The selection

operation picks out parents from the individuals whose fitness values are already received by the master node. If the master node does not wait for the final result of fitness evaluation, the master-slave PGA is said to be asynchronous, otherwise, it is synchronized. Although, this master-slave model accelerates the treatment of each generation, we cannot scale up the population to further improve the searching efficiency, since the number of individuals that can be tackled by a master node is limited.

Coarse grained PGA partitions the population into a relatively small number of demes, and each deme contains many individuals. All the GA operators are implemented concurrently on the subpopulation in each deme. One additional operator called migration is introduced for exchanging individuals between different demes. The migration operator controls the movement of individuals through several parameters, such as the topology of deme network, a migration rate restricting the number of moving individuals, a migration scheme defining the replacement policy and a migration interval determining the frequency of migrations. This coarse grained model is suitable for the computation deployed on heterogeneous computer network.

Fine grained PGA divides the population in an opposite way to coarse grained PGA. It uses a large number of computers or processors since the population is split into many demes with small subpopulations. To reduce the communication overhead caused by migration, overlapping areas are introduced into each subpopulation. The overlapping area stores the individuals that belong to more than one deme, and these individuals participate in more than single crossover and selection operations. This PGA model can be easily applied to shared memory system.

Although the traditional models illustrated above are the most popular ones for PGA, non of them is well suited to be implemented in MR framework. As it is mentioned, the master-slave model faces the scaling problem. On the other hand, the coarse and fine grained PGA need either communication between demes or shared memory, which cannot be offered by MR. Hence, we apply another relatively new parallelization method, called Dynamic Demes model, to our PGA.

Dynamic Demes PGA combines the master-slave model and the coarse-grained PGA. During the evolution in Dynamic Demes model, the whole population is treated as a single collection of individuals. After the current generation is determined in each processing cycle, the first task of the PGA is to dynamically reorganize demes, where other GA operators are independently applied. Since the information between individuals is exchanged via the dynamic reorganization, the migration operator is no longer useful, and omitted in the PGA. The reorganization of demes matches perfectly the mapping phase in MR, and other operators such selection, crossover are executed by reducers. We detail our PGA implemented by MR framework in section 4.3.2.

4.3.2 Dynamic Demes PGA in MR

At the beginning of each processing cycle, the mapper randomly regroups the entire population into r subpopulations, where r represents the number of reducers in the MR system. Each reducer takes care of N_p/r individuals, and executes GA operators

Algorithm 3 MR algorithm evaluates the initial population and finds the global minimum fitness value

```
1  class Mapper: Mapper1
2      method Map(id, G'_id)
3          for all individual ∈ G'_id do
4              Emit(rv, individual)
5
6  class Reducer: Reducer1
7      method Reduce(rv, individual)
8          LocalMin ← ∞
9          for all individual ∈ [individual] do
10             fit ← Evaluate(individual)
11             individual ← (individual,fit)
12             if fit < min then
13                 if (fit < LocalMin) then
14                     LocalMin ← fit
15         Emit(1, (LocalMin, [individual,fit]))
16
17 class Mapper: Mapper2
18     method Map(1, [LocalMin], [individual,fit])
19         for all LocalMin ∈ [LocalMin] do
20             Emit(1, LocalMin)
21
22 class Reducer: Reducer2
23     method Reduce(1, [LocalMin])
24         GlobalMin ← ∞
25         for all LocalMin ∈ [LocalMin] do
26             if LocalMin < GlobalMin then
27                 GlobalMin ← LocalMin
28         Emit(1, GlobalMin)
```

independently. When an offspring is produced, its fitness value is calculated immediately. Each reducer produces also N_p/r offspring, so that N_p offspring are produced by the whole system. When all the reducers finish producing offspring, and there is no qualified offspring, the algorithm terminates. Note that the evaluation of the initial population is separated from the main trunk of the algorithm, we illustrate the MR algorithm for the two parts in section 4.3.2.1 and 4.3.2.2 respectively.

4.3.2.1 MR for the Initial population

The individuals in the initial population are pre-generated outside of the MR algorithm, so the algorithm does not contain the selection, crossover and mutation operators. The objective of this MR is to distribute evaluation tasks and find the global minimum fitness value.

We assume that the population is stored by several chunks in Hadoop File System (HDFS). The input of the map function is the chunk *id* and the subpopulation

Algorithm 4 MR algorithm produces offspring and finds the minimum fitness value

```
1  class Mapper: Mapper1
2      method Map id, G'_{t_id}
3          for all individual ∈ G'_{t_id} do
4              Emit(rv, (individual, fit))
5
6  class Reducer: Reducer1
7      method Reduce(rv, [individual, fit])
8          LocalMin ← GlobalMin
9          for 1 to ⌈N_p/r⌉ do
10             offspring← Mutate(Crossover(Select([individual, fit])))
11             if (fit ←Evaluate(offspring))< min then
12                 offspring ← (offspring,fit)
13                 if (fit < LocalMin) then
14                     LocalMin ← fit
15         Emit(1, (LocalMin, [offspring,fit]))
16
17 class Mapper: Mapper2
18     method Map(1, [LocalMin], [individual,fit])
19         for all LocalMin ∈ [LocalMin] do
20             Emit1, LocalMin
21
22 class Reducer: Reducer2
23     method Reduce(1, [LocalMin])
24         GlobalMin ← ∞
25         for all LocalMin ∈ [LocalMin] do
26             if LocalMin < GlobalMin then
27                 GlobalMin ← LocalMin
28         Emit(1, GlobalMin)
```

G'_{id} stored in the chunk. Then, the map function extracts individuals from the subpopulation. Each individual is attached by the first mapper a random number rv whose value takes from one to the number of reducers minus one. According to rv, individuals are assigned to different reducers.

The task of the first reduce function is to evaluate the fitness value of each individual and report the local minimum fitness value in its subpopulation. Concretely, each reducer computes concurrently the fitness value of every individual. When the fitness value of an individual is obtained, it is attached at the end of each individual and compared with a local minimum fitness. If the obtained fitness value is less than the local minimum, the value of the local minimum is updated. After all the individuals are processed, each reducer outputs the local minimum fitness and the set of individuals with their fitness values in HDFS. Each part of the output data is further divided into two parts: local minimum and individuals. The two parts are stored in two different files. The former is the input of the second phase of MR, which finds the global minimum fitness value.

In the second phase of MR, the local minimum fitness of each subpopulation is gathered by the mapper. All the local minimum values are forwarded to a single reducer to calculate the global minimum fitness value. The global minimum is then regarded as the criteria for qualifying offspring.

4.3.2.2 MR for Offspring

This MR algorithm is aiming at producing, evaluating offspring, and updating the global minimum fitness value. The input of the algorithm is the current generation G_t stored in some HDFS chunks. Besides the individual itself, the input includes also its fitness value, and the global minimum fitness value. The same as Algorithm 3, Algorithm 4 contains two phases of MR.

Again, the first map function is used to regroup the subpopulations. The objective of the reorganization is not only to distribute the reduce task but also to exchange individuals in demes and prevent the algorithm to converge at a local minimum point.

The main function of the algorithm is undertaken by the first reduce phase. It is responsible for generating and qualifying offspring, and determining the local minimum fitness value. All the qualified offspring and their fitness values are written to the HDFS system with the local minimum after $\lceil N_p/r \rceil$ offspring are produced. Then, the local minimum fitnesses are send to the second MR phase to determine the global minimum fitness.

4.3.3 Complete the PGA

Let us now integrate the algorithm 3 and algorithm 4 into the sequential algorithm 2, the complete version of the PGA is illuminated in Algorithm 5.

Algorithm 5 Parallelized Genetic Algorithm for k-PCFLP

```
1  t = 0
2  GlobalMin = ∞
3  Q = ∅
4  Initialize(G_t)
5  min ← Algorithm2(G_t)
6  min ← Algorithm3(G_t, GlobalMin)
7  while Q ≠ ∅ do
8      replace(G_t,Q)
9      t = t + 1
10     Q = ∅
11     GlobalMin ← Algorithm3(G_t, GlobalMin)
12 end while
```

In Algorithm 5, the *replace* function is independent of the two MR sub-algorithms. It substitutes the worst individuals in the population with the qualified offspring in a centralized way.

4.4 Implementation Details

We implement our MR functions in Hadoop-0.20.203 released in May 2011. The functions are programed in C++ because of its efficiency as well as the need of the evaluation process. The reason will be explained later in section 4.4.1. In the following parts of this section, we elaborate our approaches to evaluate each individual and realize other GA operations.

4.4.1 Evaluating Individual

As we have mentioned before, the determination of fitness is transformed to the MCMF problem. Our implementation experience shows that the well known *Fold-Fulkerson* and *minimum mean cycle canceling* algorithms are quite inefficient, mainly because of the large size of our problem instance. Particularly, the flow graph constructed by our instance contains more than forty thousand nodes and one hundred and forty thousand arcs. Therefore we replace the optimal MCMF algorithms with the heuristic algorithms using *Scaling Push-Relabel* method proposed in [13] and [14]. Since these algorithms are implemented in C/C++, the same programming language are used to embed them.

4.4.1.1 Basic Push-Relabel Method

We use a directed graph $G = (V, E, s, t, c)$ to represent the flow graph built by our instance, where V and E are the node set and the arc set; s and t are the source and sink node; and c is a nonnegative capacity function on the arcs. The number of nodes and arcs are defined as $n = |V|$ and $m = |E|$. The graph is symmetric, arc (v, w) indicates the existence of arc (w, v). The flow from node v to node w is denoted as $f(v, w)$. The *excess* $e_f(v)$ is defined as the difference between the incoming and the outgoing flow of v. According to the conservation constraint, we know that $e_f(v) = 0, \forall v \in (V \setminus s, t)$. A preflow is a relaxed solution of the MF problem where the conservation constraints require only the excesses to be nonnegative.

The residual capacity $u_f(v, w)$ is the part of capacity of $c(v, w)$ that has not been occupied by the flow (*i.e.,* $u_f(v, w) = c(v, w) - f(v, w)$). If the residual capacity is greater than zero, the arc is residual, otherwise it is saturated. The residual graph is composed by residual arcs. The distance labeling $d : V \to \mathcal{N}$ satisfies the following conditions: $d(t) = 0$ and $\forall (v, w) \in E, d(v) \leq d(w) + 1$. If we have $d(v) = d(w) + 1$, the arc is called admissible. A node v is active if $v \notin \{s, t\}, d(v) < n$, and $e_f(v) > 0$.

The push-relabel method for MF begins with a preflow that is equal to zero on all arcs and $e_f(v)$ is zero on all nodes except s. The excess of s is set to a number that exceeds the potential flow such as the sum of capacities of all arcs out of the source plus one. Initially, $d(v)$ is the number of hops on the shortest path from v to t. Then, the algorithm repeatedly execute the following operations for all $v \in V$ and $(v, w) \in E$:

- *Push* (v, w): send $\min(e_f(v), u_f(v, w))$ units of flow from v to w if v is active and (v, w) is admissible.

- *Relabel* v: if v is active and push (v, w) dose not apply for any w, replace $d(v)$ by $min_{(v,w)\in E_f}\{d(w)\} + 1$, or by n in the case that $\nexists(v, w) \in E_f$.

The first phase of the algorithm terminates when no active node exists in the flow graph. The aim of second phase is to convert f into a flow. It is usually achieved by running the first stage backward.

To describe the push-relabel method for MC-flow, some extra notations are necessary. First of all, a real-value cost $a(v, w)$ should be associated with each arc $(v, w) \in E$. Moreover, a price function $p : V \to R$ is assigned to each node $v \in V$. The reduced cost of an arc (v,w) is $a_p(v, w) = a(v, w) + p(v) - p(w)$. For a given flow f and price p, an arc (v, w) is cost admissible if it is a residual arc of negative reduced cost. The cost admissible graph $G_{CA} = (V, E_{CA})$ is the graph induced by the cost admissible arcs. For a constant $\epsilon \geq 0$, f is said to be $\epsilon - optimal$ with respect to p, if $a_p(v, w) \geq -\epsilon$ for every residual arc (v, w).

The push-relabel algorithm maintains a flow f and a price function p, such that f is ϵ-optimal with respect to p. The initial state is $\epsilon = C$, with $p(v) = 0, \forall v \in V$, and with any feasible f having $e_f(v) = 0$. The feasible flow f can be obtained by the push-relabel algorithm for MF. Any flow is C-optimal with respect to the zero price function. The main purpose of the algorithm is to iteratively reduce ϵ by a constant factor α. After $\lceil \log_\alpha (nC) \rceil$ iterations, the algorithm terminates when $\epsilon < \frac{1}{n}$. In each iteration, f and p are refined with ϵ. The refining action is firstly to update the value of ϵ by ϵ/α; then for all arcs $(v, w) \in E$, if $a_p(v, w) < 0$, let $f(v, w) \leftarrow u(v, w)$, so that f is converted into an ϵ-optimal flow; finally, execute all the applicable push and relabel operations and return updated (ϵ, f, p). The push-relabel operation is redefined as follows:

- *Push* (v, w): send $\min(e_f(v), u_f(v, w))$ units of flow from v to w if v is active and cost admissible.
- *Relabel* v: replace $p(v)$ by $\max_{(v,w\in E_f)}\{p(w) - a(v, w) - \epsilon\}$ if v is active and not cost admissible.

The push-relabel processes in both MF and MC algorithms runs in $\mathcal{O}(n^2m)$. So the complexity of MC algorithm is $\mathcal{O}(n^2m \log(nC))$, which remains inefficient. However, in practical implementations the performance of the algorithm can be ameliorated by slightly modifying some operations as relabel and pricing.

4.4.1.2 Heuristics of Push-Relabel Method

Intuitively, the basic Push-Relabel algorithm lacks practical competence because the relabel process is a local operation. The missing global picture of distances can be repaired by the global relabeling heuristic [13]. The heuristic updates periodically the distance function by computing shortest path distances in the residual graph from all nodes to the sink. The process is realized by a backward breadth first search. The linear time approach can drastically improves the running time.

The global relabeling heuristic for MF turns to be the price update heuristic in the MC-flow context. The global price update is conducted based on the set-relabel operation introduced in [14]. Other improvements of the algorithm are brought by

price refinement and arc fixing heuristics. The main idea of the former is to decrease ϵ and not change the flow f while modifying p in an attempt to find p such that f is ϵ-optimal with respect to p. The later addresses the problem that it is not essential to examine some arcs until the optimal flow value computation, *i.e.,* if the current flow is ϵ-optimal and the absolute value of an arc cost is more than $2n\epsilon$, the push-relabel method will not change the flow on this arc.

4.4.2 Other GA Operations

Besides the evaluation scheme, other operations may also have a great impact on the overall efficient of the GA. After the practical implementation, we have refined some operations we have defined before. Specifically, we modified our population initialization, the crossover operation, and we add an easy post production stage after we obtain the result.

4.4.2.1 Initial Stage

Previously, in the initialization stage, we deploy items according to their rarenesses and popularities. Popular items should have more replicas in the individual, and rare items should be deployed less times. All the deployments are executed along with a Zipf probability distribution. However, the allocation can not guarantee that each item appears in the individual. Note that, in our instance, every item is requested by at least one user. If there is one item missing in the individual, then the flow graph generated based on the individual is infeasible and the defective is not allowed to participate the production of offspring. These defectives reduce the diversity of the initial population, and degrade the search performance of GA or even the quality of the final result.

Therefore, instead of following only the Zipf distribution, we force every item to be present in each initial individual. As it is illustrated in section **??**, each individual has 13,000 genes. We reserve the first 3,184 positions to allocate the 3,184 different items. Assuring that each item has at least one copy in the individual, we then fulfill the remain positions based on items' popularity. To further increase the diversity of individuals, we introduce an exchange stage after each individual is replenished. For the first 3,184 gene, we randomly change their positions with other genes.

4.4.2.2 Crossover, Mutation and Post Production

In order to prevent the born of deformities, the primary task of the crossover is also to guarantee that the unique gene in the parents still exists in the offspring. Therefore, at the beginning of the crossover operation we traverse all genes in both parents and pick out unique ones. Then, we start the merge operation for each section. Those unique genes have the highest priority to be allocated in the offspring. Thereafter, the crossover operation inserts the genes that appear in both parents into the section of the offspring. The rationale behind is that, intuitively, if the same item presents in both qualified parents, the item has a larger probability to be required by the user in the region. Finally, it fills the rest positions by randomly selecting other genes that exist in the section of only one parent.

Originally, we said that the mutation operation is reserved for the gene "0", which means an empty storage space. During the mutation operation, each gene has a very low probability to mutate to zero. But the implementation reveals that it is not necessary to give the privilege to "0" gene. Because we no longer consider the assignment cost but only the delivery cost. An other reason is that the mutation may produce unqualified offspring by mutating unique gene. Moreover, we will optimize the utilization of storage space by an easy post production stage. So the mutation operation is now an exchange action. Each gene has a small chance to change its position with another gene.

The post production stage is yielded based on the observation that several replicas of a same item may exist in one section. This is provoked by the mis-treatment in the crossover operation, where replicated item has larger probability to be allocated, and also the mutation operation, where exchange is implied with no condition. Again, since we do not consider the assignment cost, the replicas do not impact the quality of the final result even if the replication means the waste of storage capacity in practice. Therefore, after the algorithm converges to the optimal solution, we execute the post production where the appearance of an item in each section is restricted to be one. Unwanted genes are replaced by zeros meaning that redundant replicas of an item are eliminated.

4.5 First Result

In this section, we illustrated the instance used to test our PGA, and then give the first result obtained.

4.5.1 Instance

The PGA described above has been tested on an single node cluster of MR with 4 ADM 2.4Hz processors and 2 GB memory. In our implementation we have used two different categories of trace. One category comes from the real download trace of Orange VoD service from March 13 to April 2, 2011. The other category is the list of videos obtained by the item similarity recommendation algorithm. Specifically, the algorithm takes the records in the first two weeks of the real trace, called warm-up part, and then it recommends for each user some videos to be requested in the coming week. Every user is recommended with 3 to 10 videos. The function of the recommendation algorithm is out of scope of this report. The structures of records in the two categories are given in figure 4.1 and 4.2. In the trace record, the first field is the identification of a user. The second field is the number of the video or clip that are downloaded by the user. The field region indicating the geographical location of the user and the last one shows the time when the video was requested.

user ID	*item ID*	*region*	*timestamp*

Figure 4.1: Structure of trace record

user ID	*item ID*	*region*

Figure 4.2: Structure of recommendation

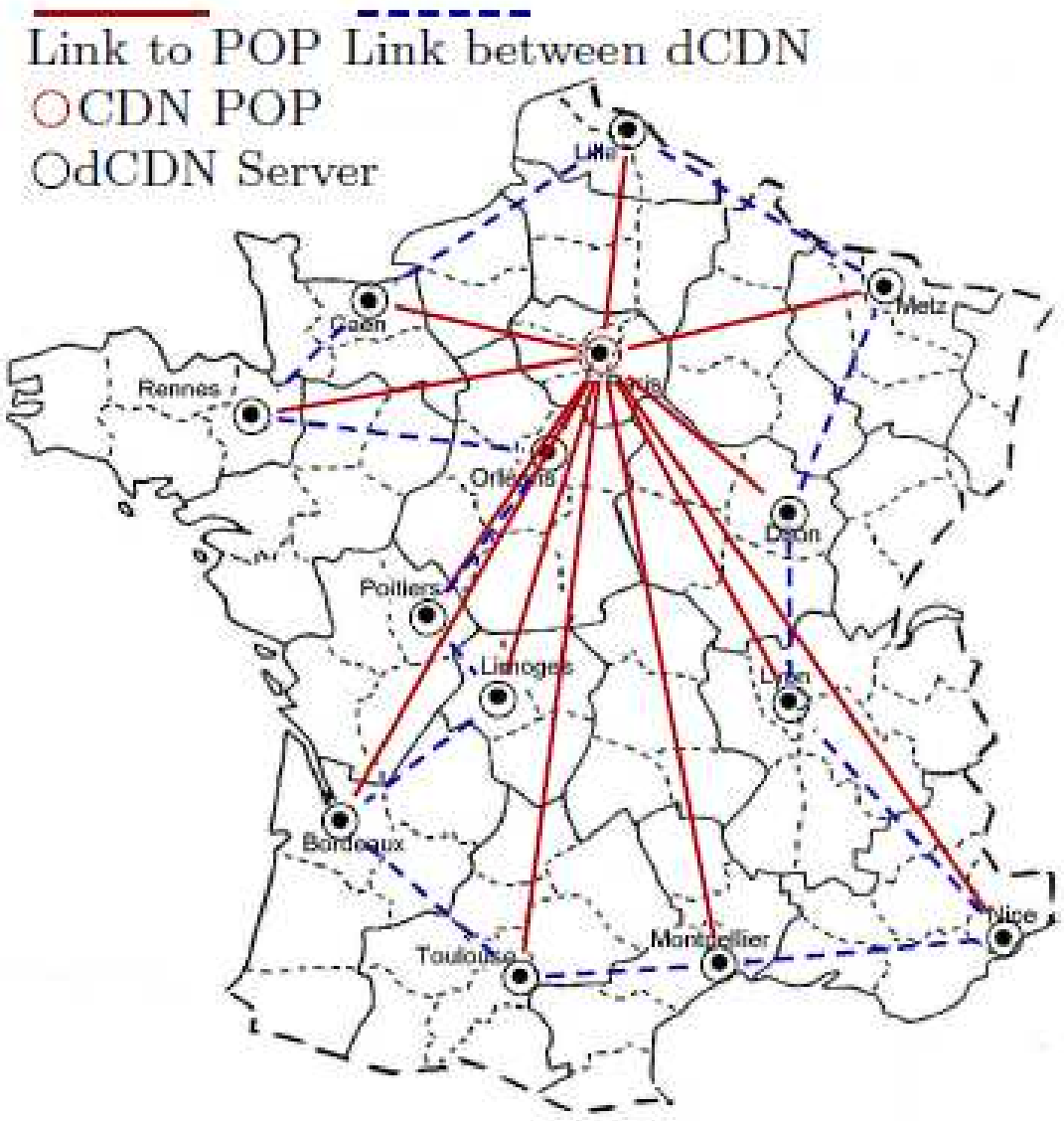


Figure 4.3: Topology of the instance

Note in the recommendation, the algorithm is able to predict only the videos that are potentially required by users but not the time when these videos will be downloaded. So there is no field manifesting the requested time of each video. Other fields are the same as they are in the trace record. Both categories involve the requests from 3,516 different users. Since there are 23,806 records in the recommendation, we take the same quantity of download records in the last week of the trace, and call it as the test part. The diversities of videos in recommendation and test part are a bit different. The recommendation contains 3,184 distinct videos, and the real trace has 3686 ones respectively.

(1.5,1.6)

Users scatter in 13 regions in France. Each region is equipped with a dCDN server. The servers are assumed to be homogeneous with the storage capacity of

1,000 videos and the service bandwidth of 3,000 users. The topology of the dCDN network is described in figure 4.3. The red circle represents the CDN server located at the Point of Presence (PoP) in Paris. We suppose that each dCDN server is connected directly with the CDN server so that they can easily retrieve the original copy of video. These connections are regarded as the links toward CDN provider or transit networks. Therefore, traffics transfered through red links will be charged either by CDN provider or up tier network provider, and generate expensive cost for an ISP. On the other hand, we allow a lightweight cooperation between dCDN servers. Concretely, each pair of geographically adjacent dCDN servers is connected by a peering link indicated by dashed blue line in the figure. Moreover, we assume that a dCDN server keeps a list of videos stored on the other dCDN servers which are closer to it than the CDN server, so that it is able to find the nearest replica of required video. So each server may have one or more servers that cooperate with it, and we call this group of servers as cooperation group. For example, the cooperation group of server at Rennes includes servers at Caen and Orleans, and the cooperation group of server at Nice includes servers at Toulouse, Montpellier, Lyon and Dijon. Peering links are intra-domain links managed by the ISP. Thus, the ISP does not need to pay other communities for the traffic transfered on these blue links. However, we still assign some cost on them representing the management overhead. Specifically, we assume that the cost for transferring a unit of data on a blue link is a half of the cost on a red one.

The metric used to calculate the cost is the geographical distance between servers multiplied by the bandwidth consumed by video streaming. The cost generated by the traffic on blue links are further divided by two. Since we assume that all videos have the same playback bit-rate, we consider only the distance between servers.

4.5.2 Measurements

To investigate the benefit yielded by our PGA, we compare it with the solutions provided by other resource allocation schemes including centralized service, random allocation. The **centralized service** is taken as a criteria where all the requests are satisfied by the CDN PoP. In the following paragraphs, we will illustrate how the random allocation and our PGA solution are implemented.

Let us start with the random allocation. We have tested four different configurations in our experiments, namely, *i)* random allocation treating recommendations without cooperation (between servers); *ii)* treating recommendations with cooperation; *iii)* treating real trace without cooperation and *v)* treating real trace with cooperation. In the first configuration, we extract the 3,184 distinct videos in the recommendations. Then each unit of storage space is randomly filled by one video until all the thirteen servers are fulfilled. Thereafter, we counter for each region the number of recommendation records that cannot be satisfied by the corresponding regional server. So the cost produced by a region is the number of unsatisfied requests multiplied by the distance between the server and PoP. The total cost is the sum of the cost yielded by the thirteen regions.

In the second configuration, the allocation process is the same as it is in the first one. The difference lies in the cost computation procedure. Once a request

cannot be satisfied by the regional server, the required video will be searched in the cooperation group. If it exists in the cooperation group, the cost to serve this request is half of the distance from the regional server to the cooperating server. Otherwise, the video should be retrieve from the CDN server which generates the cost as the distance between the CDN and the regional server.

The cost computation procedure of the third and the forth configuration is the same as the first and the second one respectively. However, since we intend to handle real trace this time, the video set is extracted from the logs in the warm-up part.

The setup of the experiments for our PGA is simpler than the random allocation. We select the best individual from the solutions given by the PGA, and deploy the videos according to the individual. The warm-up part is not used in the placement of videos. However, it is not useless since the recommendations are generated based on it. Same configurations are investigated. Please remind that the performance of the PGA handling the real trace is highly depend on the accuracy of the recommendation algorithm. So the results given by the configurations where PGA is used to serve real trace may not be good references of the performance.

4.6 Results

This section compares the cost generated by different allocation schemes. Instead of display the absolute value of cost, we show the ratio of the cost produced by a scheme to the cost yielded by the criteria, namely, the centralized service scheme in Figure 4.4.

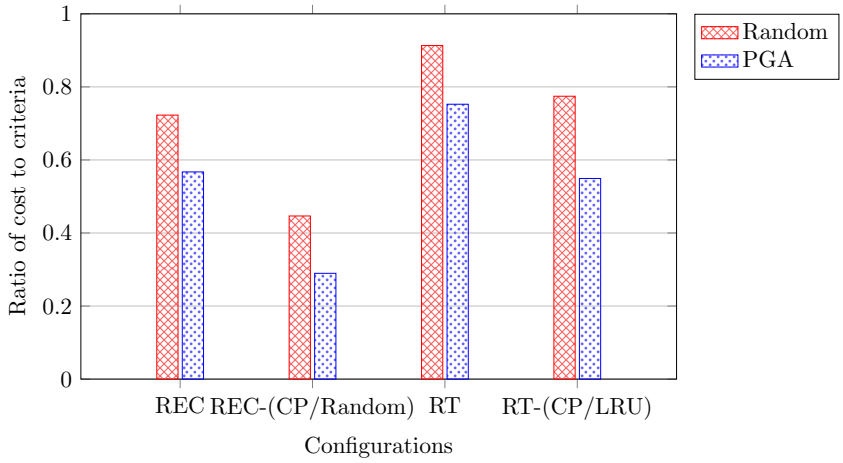


Figure 4.4: Cost produced by different chunk placement schemes.

In the figure, the title REC and RT means the cost of handling recommendations and real trace respectively. The abbreviation CP indicates the cooperation between dCDN servers. The word Random and LRU means the two different replacement policy used by the LRU scheme. The cost ratios of the random and proportional schemes are the average values of 100 runs.

In the treatment of recommendations, the performance of the random allocation is quite impressive when servers cooperate. It reduces more than 50% of the cost

comparing with the criteria. In the same configuration, our optimal placement outperforms random allocation about 35%, thus, its total gain is a bit more than 70%.

In the treatment of real trace, the random allocation shows its weakness. Even in the cooperative case, it diminishes the cost by only 20%. When there is no cooperation between servers, the cost generated by the random allocation is almost the same as the centralized scheme. The performance our optimal placement degrades to certain degree. The degradation is comprehensible. The optimal placement gives worse results in handling real trace because the recommendation algorithm is not accuracy. In fact, the precision of the recommendation for each user is about 10%. That is, only one out of the ten recommended videos will be required in reality. But its performance is not so bad as the random one due to the aggregation of requests in each region.

Besides the cost, the statistic of requests served by dCDN servers are listed in Table 4.1. These numbers are consistent with the cost signified in Figure 4.4.

4.7 Future Work

Since the PGA is successfully implemented in MR, the next step is to test different configurations to address the most efficient configuration for finding the optimal allocation. At the same time, we can compare the performance of caching and pushing delivery schemes.

schemes / configurations	Random	PGA
REC	6409	10297
REC-(CP/Random)	15823	19757
RT	2060	5847
RT-(CP/LRU)	6540	12820

Table 4.1: Number of requests served by dCDN servers

5 Distributed replication and caching strategy for ViPeeR

5.1 Introduction

The rise of popularity of video streaming services has resulted in increased volumes of network traffic that, in turn, has created bottlenecks in the networks causing degradations of the perceived quality. Early, in-network caching was proposed as a mean to get the contents closer to the end-users [16]. With the shift towards information-centric networking (CCN), this logic is pushed further [25]. In fact, the CCN paradigm focuses on the data itself, rather than focusing on having a reference to its physical location. The resulted communication scheme is, thus, no longer end-to-end data delivery as in the current Internet architecture. Particularly, CCN introduces two distinct techniques: contents caching and replication [28]. Contents caching mainly addresses the contentsÕ management in a particular cache, while content replication consists in disseminating data in its way to the destination. However, one should consider the mutual impact existing between these two techniques. Indeed, the benefits of contentsÕ replication can be completely cancelled with an inappropriate caching strategy.

One of the best ways to tackle the congestion, and particularly peering links congestion, is to achieve high cache hit ratio by making the contents to be requested available inside the intra-domain.

In this chapter, we intend to propose a fully distributed solution, which is based on the architecture proposed in deliverable $D1.3$. The proposed approach is conceived with the main aim to reduce the average cache miss rate (in the intra-domain) of the content hosted within the CDN. To achieve this goal, we propose to address at the same time: the minimization of duplicate contents within the intra-domain while making the popular content closer to the end-users.

5.2 Overview of the network architecture

In contrast with the BitTorrent approach, which is considered in the iCode architecture [15], the proposed dCDN "Distributed CDN" architecture for media delivery is based on a topology-aware P2P overlay network, which is managed by the Network Operator (NO). Considering the network topology clearly allows optimizing resourcesÕ usage when exchanging data (i.e. chunks). In fact, this allows to retrieve data from the nearest cache, without any support of a monitoring system. Moreover,

in contrast with end-to-end approaches (e.g. BitTorrent and P4P), this allows to seamlessly and efficiently support possible replication strategies, which is considered as a key element for the overall network efficiency [28].

The proposed architecture, which is described in 5.1, comprises four major elements: the VECs "ViPeeR Edge Caching", the VCCs "ViPeeR Core Caching", the VPPs "ViPeeR Point of Presence" and the dTracket "Distributed Tracker".

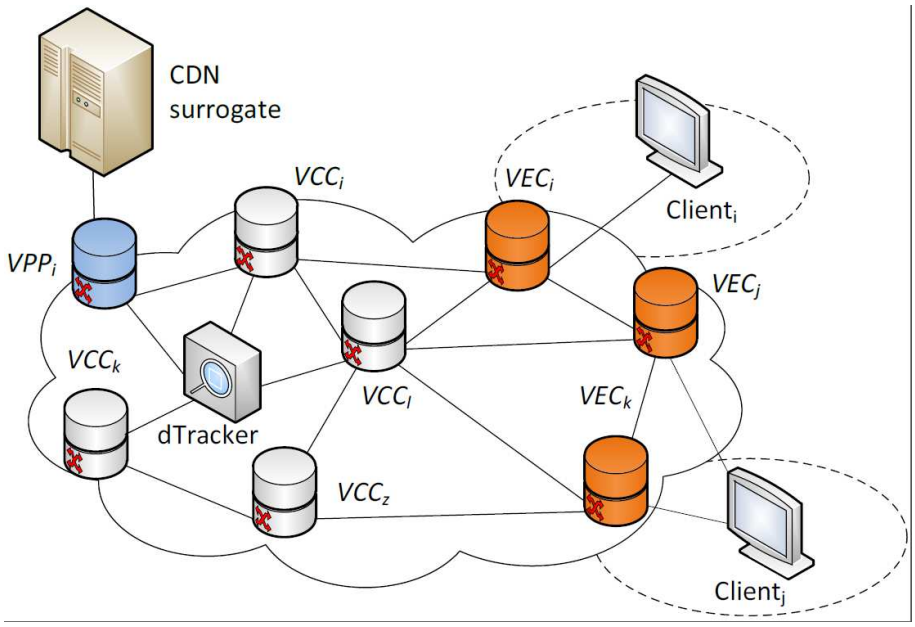


Figure 5.1: Network elements

dTracker the dTracker is in charge of modifying the MPD to point-out to the elements of the intra-domain and more precisely the optimal[1] VEC to a particular client.

VECs the VECs, which represents the entry points of the network, don't necessarily contain the requested chunks. This allows being completely independent from the caching strategies, while supporting any type of clients with the only requirement that the manifest is modified by the dTracker. The VECs are in charge of retrieving (reactively or proactively) data and sent it out to the end-user.

VCCs No major difference exists between the VCCs and VECs. The only difference is that the VECs are in charge of receiving the requests from the end-users, thanks to the dTracker action.

VPPs When the content is not present in the intra-domain the VPPs are the network elements in charge of retrieving the content directly from the CDN.

[1]It is meant by optimal the nearest VECs geographically or the VECs optimizing resourcesÕ use

5.3 Proposed combined content replication and caching technique

Caching and replication strategies are the two keys of efficient resources management in the future information-centric networks. Indeed, in these networks, the content, which is split into several chunks, is replicated into the different elements of the network during its transportation towards the destination. It is, then cached locally according to predefined caching policy.

An efficient replication strategy should guarantee that a maximal number of chunks can always be reached within the intra-domain (i.e. minimization of replicas). The rational is to minimize the distance (i.e. the amount of used resources and the latency) between the end-users and the contents to be requested while reducing significantly the peering links' load (i.e. minimizing content retrieval from the CDN). Particularly, one should consider the popularity of the contents as one of the main criteria in the replication technique since it determines the biggest portion of the clients requests' rate (see [7] and [8] for more details).

In the following, we describe the two main functions describing the behavior of the proposed protocol. The first one consists in the reception of the request for a content event and the second one consists in the data reception event (i.e. a response to the request). These two functions aim to achieve the objectives described above. Note that the proposed approach modifies slightly the actual definition of CCN, as described in [25].

5.3.1 Reception of a request event

The algorithm 6 describes the network elements behavior when receiving a request for a content. Note that the node behavior is similar to CCN node behavior excepting the request structure, which comprises an additional field, named *RepProb*, helping in replicating wisely the content to be send to the end-users. In fact, *RepProb* represents the replication probability, which depends on the popularity of the content. Note that the functions used in the algorithm are defined in table 5.3.1.

5.3.2 Description of the receive data event

The algorithm 5.3.2 describes the network elements behavior when receiving data. Note that the functions used in the algorithm are defined in table 5.3.1.

5.4 Conclusion

This section focus on a preliminary description of a combined caching and replication technique, which will be considered for the distributed scenario of the ViPeeR architecture.

Algorithm 6 Description of the receive request event

Require: VCC_i: Network element; *Req*: Received request; *Resp*: Structure embedding the content and the RepProb;

```
1  if RcvReq(VCC_i, Req) then
2      ContentName ⇐ Req.ContentName
3      if CheckCS(ContentName) then
4          Resp.ContentName ⇐ ContentName
5          Resp.RepProb ⇐ CalcProb(ContentName)
6          ForwardC(VCC_i, Resp)
7      else
8          if AlreadyReq(ContentName) then
9              AddIF(VCC_i, ContentName)
10         else
11             if Discover(ContentName) then
12                 AddIF(VCC_i, ContentName)
13                 ForwardReq(GetDest(ContentName), VCC_i, Req)
14             else
15                 DeleteR(Req)
16             end if
17         end if
18     end if
19 end if
```

Algorithm 7 Description of the receive data event

```
1  if RcvData(VCC_i, Resp) then
2      ContentName ⇐ Resp.ContentName
3      if CheckCS(ContentName) then
4          DeleteR(Resp)
5      else
6          if AvailableSpace(Resp.Content) then
7              Cache(Resp.Content)
8              Resp.Cached ⇐ True
9              UpdateProba(Resp.RepProb, True)
10         else
11             Rnd ⇐ GetRandom()
12             if Rnd < Resp.RepProb then
13                 Cache(Resp.Content)
14                 Resp.Cached ⇐ True
15                 UpdateProba(Resp.RepProb, True)
16             else
17                 UpdateProba(Resp.RepProb, False)
18             end if
19         end if
20         ForwardC(GetRequesters(ContentName), Resp)
21     end if
22 end if
```

Table 5.1: Description of the functions

Function	**Description**
$RcvReq(VCC_i, Req)$	Function checking the correctness of the request Req received from VCC_i
$CheckCS(ContentName)$	Check the content store whether $ContentName$ is present locally
$CalcProb(ContentName)$	Calculate the replication probability for $ContentName$
$ForwardC(VCC_i, Resp)$	Forward the content corresponding to $ContentName$ to VCC_i
$AlreadyReq(ContentName)$	Returns $True$ if $ContentName$ was already requested
$AddIF(VCC_i, ContentName)$	Add the interface VCC_i as a requester for the content $ContentName$
$Discover(ContentName)$	Returns $True$ if a reactive technique is used or a path to the requested content exists in case of proactive technique
$GetDest(ContentName)$	Returns the list of the possible destinations for $ContentName$
$ForwardReq(L, VCC_i, Req)$	Forward the request Req to the list of elements L excepting the originating node VCC_i
$DeleteR(R)$	Stop the propagation of the request/response R
$RcvData(VCC_i, Resp)$	Function checking the correctness of the answer $Resp$ received from VCC_i
$AvailableSpace(C)$	Check if there's enough space to cache the content C
$Cache(C)$	Cache the content C following the selected caching policy
$UpdateProba(P, B)$	Update the replication probability P
$GetRandom()$	Get a random number
$GetRequesters(ContentName)$	Get the list of requesters for the content $ContentName$

6 Conclusion

To conclude this deliverable, we have provide an solution for co-locating Hadoop clusters and will be available as a contribution on the Apache Hadoop community. In parallel, the genetic algorithm had implemented the MapReduce framework and the next step will be to test differents configurations in order to have the optimal allocation of contents. And also, we will continue on the distributed replication and caching strategy for ViPeeR.

Bibliography

[1] Aws elastic compute cloud. http://aws.amazon.com/ec2/.

[2] Hadoop: Open source implementation of mapreduce. http://hadoop.apache.org/.

[3] Yarn. http://hadoop.apache.org/common/docs/r0.23.0/.

[4] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-Parallel Computing. In Proc. of NSDI, 2012.

[5] M. W. Berry. Large scale sparse singular value computations. International Journal of Supercomputer Applications, 6:13–49, 1992.

[6] M. Brand. Fast online svd revisions for lightweight recommender systems. In In SIAM International Conference on Data Mining, 2003.

[7] Y. Carlinet, T. D. Huynh, B. Kauffmann, F. Mathieu, L. Noirie, and S. Tixeuil. Four Months in DailyMotion: Dissecting User Video Requests. In TRAC 2012 - 3rd International Workshop on TRaffic Analysis and Classification, Limassol, Chypre, Aug. 2012.

[8] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. Analyzing the video popularity characteristics of large-scale user generated content systems. IEEE/ACM Trans. Netw., 17(5):1357–1370, Oct. 2009.

[9] H. Chang, M. Kodialam, R. R. Kompella, T. V. Lakshman, M. Lee, and S. Mukherjee. Scheduling in MapReduce-like Systems for Fast Completion Time. In Proc. of IEEE INFOCOM, 2011.

[10] Y. Chen. We don't know enough to make a big data benchmark suite - an academia-industry view. In Proc. of Workshop on Big Data Benchmarking, 2012.

[11] Y. Chen, S. Alspaugh, and R. Katz. Interactive query processing in big data systems: A cross-industry study of mapreduce workloads. In Proc. of VLDB, 2012.

[12] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In Proc. of IEEE Mascots, 2011.

[13] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. Integer Programming and Combinatorial Optimization, 920:157–171, 1995.

[14] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. Integer Programming and Combinatorial Optimization, 920:157–171, 1995.

[15] K. Cho, H. Jung, M. Lee, D. Ko, T. T. Kwon, and Y. Choi. How can an isp merge with a cdn? IEEE Communications Magazine, 49(10):156–162, 2011.

[16] J. Choi, J. Han, E. Cho, T. Kwon, and Y. Choi. A Survey on content-oriented networking for efficient content delivery. Communications Magazine, IEEE, 49(3):121–127, Mar. 2011.

[17] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In Proc. of NSDI, 2012.

[18] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In Proc. of OSDI, 2004.

[19] E. Friedman and S. Henderson. Fairness and efficiency in web server protocols. In Proc. of Sigmetrics, 2003.

[20] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resources types. In Proc. of NSDI, 2011.

[21] M. Harchol-Balter. Queueing disciplines. In Wiley Encyclopedia Of Operations Research and Management Science. John Wiley & Sons, 2009.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In Proc. of NSDI, 2011.

[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In Proc. of ACM EuroSys, 2007.

[24] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In Proc. of SOSP, 2009.

[25] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard. Networking named content. Communications of the ACM, 55(1):117–124, Jan. 2012.

[26] K. Kc and K. Anyanwu. Scheduling Hadoop jobs to meet deadlines. In Proc. of CloudCom, 2010.

[27] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, 42(8):30–37, Aug. 2009.

[28] C.-A. La, P. Michiardi, C. Casetti, C.-F. Chiasserini, and M. Fiore. Content Replication in Mobile Networks. IEEE Journal on Selected Areas in Communications, 2011.

[29] P. Resnick, N. Iacovou, M. Sushak, P. Bergstrom, and J. Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In 1994 ACM Conference on Computer Supported Collaborative Work Conference, pages 175–186, Chapel Hill, NC, 10/1994 1994. Association of Computing Machinery, Association of Computing Machinery.

[30] T. Sandholm and K. Lai. Mapreduce optimization using regulated dynamic prioritization. In Proc. of Sigmetrics, 2009.

[31] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In Proc. of Workshop on Job Scheduling Strategies for Parallel Processing, 2010.

[32] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In Proceedings of the 10th international conference on World Wide Web, WWW '01, pages 285–295, New York, NY, USA, 2001. ACM.

[33] J. Tan, X. Meng, and L. Zhang. Performance analysis of Coupling Scheduler for MapReduce/Hadoop. In Proc. of IEEE INFOCOM, 2012.

[34] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In Proc. of ICAC, 2011.

[35] A. Verma, L. Cherkasova, and R. H. Campbell. Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance. In Proc. of Mascots, 2012.

[36] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin. FLEX: A slot allocation scheduling optimizer for MapReduce workloads. In Proc. of International Middleware Conference, 2010.

[37] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proc. of ACM EuroSys, 2010.

[38] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions. In Proc. of NSDI, 2012.