Programme ANR VERSO

Projet VIPEER

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

Décision n° 2009 VERSO 014 01 à 06 du 22 décembre 2009
T0 administratif = 15 Novembre 2009
T0 technique = 1er Janvier 2010

Deliverable 4.5

Report on CDN/dCDN performance evaluation and analysis

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

Edited by:
J. Garnier (NDS Technologies France)

February 2013

Telecom Bretagne; Eurecom; INRIA; NDS Technologies France (now part of Cisco)

Abstract

This document aims to present the report on CDN/dCDN performance evaluation and analysis. We already exposed the evaluation of the Prefetching on the previous deliverable, that's why we will not have any update on this report. Regarding the Hadoop Fair Sojourn Protocol, we will detail a refined version and expose the next steps for having a production-ready solution. After this, we will describe what are the benefits of operating a dCDN for the operator and also for the end user.

Keywords: Hadoop, Parallel Processing, MapReduce, dCDN, ISP, NF-DASH

Contents

1 Preface 5
1.1 Purpose of this document . . . 5
1.2 Referenced ViPeeR deliverables . . . 5

2 Parallel Processing in ViPeeR: Scheduling Problems 7
2.1 Introduction . . . 7
2.2 Hadoop Fair Sojourn Protocol . . . 9
2.2.1 The Job Scheduler . . . 9
2.2.2 Job size estimation and training . . . 11
2.2.3 Job Preemption . . . 13
2.3 Experiments . . . 15
2.4 Conclusion . . . 15

3 Benefits of operating a dCDN 17
3.1 Evaluation of Workload and Overall Cost . . . 17
3.1.1 Simulation Setup . . . 17
3.1.2 Measurements . . . 18
3.1.3 Simulation Results . . . 20
3.1.4 Prototyping NF-DASH . . . 23
3.2 Evaluation of Impact on Infrastructure . . . 24
3.2.1 Results . . . 25

4 Conclusion 31

32

Bibliography 33

1 Preface

1.1 Purpose of this document

This document aims to present the report on CDN/dCDN performance evaluation and analysis. We already exposed the evaluation of the Prefetching on the previous deliverable, that's why we will not have any update on this report. Regarding the Hadoop Fair Sojourn Protocol, we will detail a refined version and expose the next steps for having a production-ready solution. After this, we will describe what are the benefits of operating a dCDN for the operator and also for the end user.

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
4.	D4.4	Preliminary report on CDN/dCDN modeling and analysis

2 Parallel Processing in ViPeeR: Scheduling Problems

This Chapter is dedicated to a refined version of the size-based scheduler introduced in D.4.4. For an introduction and background information about scheduling problems in general and size-based schedulers in particular, we refer to Chapter 3 of D.4.4. In what follows, we report our work on a complete re-design of the scheduler component.

2.1 Introduction

The advent of large-scale data analytics, fostered by parallel processing frameworks such as MapReduce [6], has created the need to organize and manage the resources of clusters of computers that operate in a shared, multi-tenant environment. For example, 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. Initially designed for few and very large batch processing jobs, data-intensive scalable computing frameworks such as MapReduce are nowadays used by many companies for production, recurrent and even experimental data analysis jobs. This is substantiated by recent studies [4, 10] that analyze a variety of production-level workloads (both in the industry and in academia): an important characteristic that emerges from such works is that there exists a stringent need for *interactivity*. The number of small jobs might be dominant in current workloads: these are preliminary data analysis tasks involving a human in the loop, which for example seeks at tuning algorithm parameters with a trial-and-error process, or even small jobs that are part of orchestration frameworks whose goal is to launch other jobs according to a work-flow schedule.

In this work, we study the problem of job *scheduling*, that is how to allocate the 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]. In addition to the default, first-in-first-out (FIFO) scheduler implemented in Hadoop, recently, several alternatives [13, 3, 8, 9, 11, 12] have been proposed to enhance scheduling: in general, existing approaches aim at two key objectives, namely *fairness* among jobs and *performance*. Our key observation is that fairness and performance are non-conflicting goals, hence there is no reason to focus solely on one or the other objective. Furthermore, we revisit the notion of scheduling performance and propose to focus on job *sojourn time*, which measures the time a job spends in

the system waiting to be served and its execution time. Short sojourn times cater to the interactivity requirements discussed above.

We thus proceed with the design and implementation of a new scheduling protocol that caters *both* to a fair *and* efficient utilization of the cluster resources. Our solution, called Hadoop Fair Sojourn Protocol (HFSP) belongs to the category of size-based, preemptive scheduling disciplines. In addition to addressing the problem of scheduling jobs characterized by a complex structure in a multi-processor system, we propose an efficient method to implement size-based scheduling when job size is not known a-priori. Essentially, HFSP allocates cluster resources such that job size information is inferred while the job makes progress toward its completion. The scheduling discipline benefits from preemption to achieve short job sojourn times; however, preemption is not readily available in Hadoop. As such, we introduce a new set of primitives that enables HFSP to interrupt and eventually resume running jobs, and show in which cases this approach is superior to the widely adopted technique of killing running tasks to make room for other jobs.

The contribution of our work can be summarized as follows:

- We design and implement the system architecture of HFSP, including a (pluggable) component to estimate job sizes, a dynamic resource allocation mechanism that strives at efficient cluster utilization and a new set of low-level primitives that allow preemptive disciplines. HFSP is available as an open-source project.

- We design and implement a new scheduling discipline inspired by the Fair Sojourn Protocol [7], which operates in a multi-processor context and that caters to short job sojourn times, when compared to widely used alternatives such as FIFO and processor-sharing schedulers. One of the main consequences of the HFSP discipline is that small jobs, for which "interactivity" is important, do not wait for a long time before being awarded cluster resources. The HFSP scheduler is also beneficial to medium-large jobs which are granted a large fraction of cluster resources.

- We perform an extensive experiment campaign, where we compare the HFSP scheduler to the two main schedulers used in production-level Hadoop deployments, namely the FIFO and the Fair schedulers. For the experiments, we use (and contribute to their further development) state-of-the-art workload suite generators that take as input realistic workload traces. In addition we contribute to the development of the standard Hadoop emulator [1], which we use in conjunction to a large cluster deployed in the Amazon elastic computing cloud. Our results show that the average sojourn time achieved by the jobs of our workload is drastically reduced with respect to the other scheduler we examined. In addition, we show results that substantiate the claim of an efficient cluster resource utilization under heavy loads.

2.2 Hadoop Fair Sojourn Protocol

The design and implementation of a new scheduling component for Hadoop is a delicate task, as scheduling and resource allocation decisions determine, to a large extent, job performance. In the following we highlight the key problems we address in this work, namely: the design of the scheduling algorithm for a multi-processor system (cf. Sect. 2.2.1), an on-line mechanism to estimate job size without "wasting" resources (cf. Sect. 2.2.2) and finally a set of new primitives to suspend and resume jobs, which is a requirement for preemptive scheduling disciplines (cf. Sect. 2.2.3).

2.2.1 The Job Scheduler

The original FSP discipline – which inspires HFSP – is designed for a single-server system, in which jobs have a simple structure. Extending the concepts of FSP to work in a multi-processor system is not trivial. MapReduce jobs have a complex structure, with temporal dependencies among tasks. Moreover, the discrete nature of compute slots to execute jobs affects how *job aging* – that tracks how much work has been done for each job in the system – is computed. In addition, HFSP requires an appropriate definition of job size, which can handle jobs with different "shapes" (e.g., 100 tasks of 1 s vs. 2 tasks of 50 s). Finally, data locality – that is, making sure that tasks operate on local data – requires special care in taking scheduling decisions.

We now describe the HFSP scheduling algorithm in detail. Let's assume, for now, job sizes to be known and focus on the issues that arise in a multi-processor setting. Then, we'll describe the complete operation of HFSP, including the process of estimating job sizes. We remark 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. (cf. Sect. 2.2.2).

The scheduling algorithm is divided in two parts. The first executes every time a new job arrives or whenever a task or a job completes; the HFSP algorithm "simulates" what would happen if the scheduler was to behave as processor sharing, 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. The second part executes when a free compute slot is available in the cluster. Such slot is scheduled to execute a task of the first job in the list of jobs sorted by projected finish time in ascending order. Scheduling task execution is conditioned by *data locality*, as explained later.

The virtual cluster. HFSP uses a virtual cluster to simulate a processor sharing scheduling discipline. The virtual cluster simulates the same resources available in the real cluster: it has the same number of machines and the same configuration of slots (Map or Reduce) per machine. When a job arrives in the system, the virtual cluster uses its (estimated) size to represent the amount of remaining work that job needs to do, which can be used to compute the projected finish time. It is fundamental to notice that in this work the size of a job is expressed in a "serialized" form, that is the sum of the runtime of each of its tasks, as if they were to be executed

in series on a single slot (cf. Sect. 2.2.2). As a consequence, the remaining amount of work of a job is *independent* of the resources available in the cluster. This choice simplifies the design of a job aging function and mitigates the impact of failures in the underlying cluster. Next, we describe how *resource allocation* in the virtual cluster and *job aging* work.

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 (in terms of their number of tasks). As such, small jobs are implicitly given priority in the virtual cluster, which reinforces the idea of scheduling small jobs as soon as possible.

Job aging. The HFSP algorithm keeps track of, in the virtual cluster, the amount of work done by each job in the system. Each job arrival or task/job completion triggers a call to the job aging function, which uses the time difference between two consecutive events as a basis to distribute progress among each job currently scheduled in the virtual cluster. In practice, each running task in the virtual cluster makes a progress corresponding to the above time interval. Hence, the "serialized" representation of the remaining amount of work for the job is updated by subtracting the sum of the progress of all its running tasks in the virtual cluster.

Data locality. When assigning a new task to a free slot in the cluster, the HFSP algorithm uses the principle of delay scheduling: it first checks whether the tasks has local data to operate on; if data locality on such slot is not possible, the scheduler waits for another slot to become available. After a number of delayed task assignments (in practice, we use the same timeout mechanism used in the original delay scheduler [13]), the scheduler finally allocates a slot to the task. Note that unused slots for a non-local task are assigned to other jobs. The discussion above clearly applies to Map tasks, as in general there is no data locality for Reduce tasks. As explained in Sect. 2.2.3, HFSP implements a preemptive scheduling discipline, which calls for a mechanism to handle Reduce tasks data locality as well.

HFSP: complete operation HFSP is built as a hierarchical scheduler in which a **top-level scheduler** implements a dynamic resource allocation mechanism to provision cluster resources to the **job scheduler** (described above) and the component (similar in nature to a scheduler) used to estimate job sizes, that we call the **Training module** (cf. Sect. 2.2.2). Indeed, job size information is not available in practice. As such, the top-level scheduler aims at minimizing the delay (which contributes to sojourn times) required to proceed with job size estimation. In addition, the top-level scheduler also strives to avoid adding "waves" to a job, which could be introduced by job size estimation and that contribute to larger sojourn times.

When a job arrives in the system, the **job scheduler** uses an arbitrary size to proceed with its operation. In this work, the initial estimate for a Map phase size[1] is the number of tasks (each corresponding to an HDFS block) times the average duration of recently executed Map tasks of other jobs. The initial estimate is weighted by a configurable "confidence" parameter ξ, that takes values in the range

[1]The discussion is similar for the Reduce phase.

$[1, \infty]$. At one extreme (values close to 1), the initial estimate of job size is heavily influenced by recently executed jobs; at the other, the job scheduler assigns a low priority to the job, due to an infinite size, and does not allocate resources to its tasks. Thus, the training delay depends on ξ: small values imply that cluster resources are rapidly provisioned for a new job; large values imply that jobs are granted resources only when their size estimation completes. We note that an under-estimated job size could involve job preemption, which might contribute to larger sojourn times. In our experiments, which we execute using synthetic jobs with no skew in size distributions, we use $\xi = 1$. Note also that, in general, training delay primarily affects small jobs that do not complete in the estimation phase.

The top-level scheduler responds to the arrival of a new job by allocating a given set of resource to the **Training module**. Indeed, the first tasks of a new job are scheduled by this module to proceed with the job size estimation. The remaining tasks – if any – are handled by the job scheduler, which operates according to the initial job size estimate described above. Once a more accurate estimate of the job size is available, the job scheduler *updates* the remaining amount of work to be done for the job and operates according to the new job size.

In summary, the top-level scheduler strives at balancing resource allocation between job size training and size-based scheduling. A job obtains at least a fair[2] share of resources required for size estimation, and, in addition, a number of slots that depend on the initial job size estimate. The job scheduler eventually reassess the amount of resources each job receives based on new estimates of job sizes. The consequence of this design is that inaccurate initial estimates do not cause great damage to cluster resource allocation. Next, we delve into the details of how to compute job size estimate and on job preemption.

2.2.2 Job size estimation and training

HFSP uses a **Training module** to produce estimates of job sizes, which the scheduler then uses to track the amount of work each job needs to do. The training module uses a pluggable *estimator* to output job size estimates. When a new job arrives in the system, the Training module executes a fraction of its tasks (that we label the *sample set*): while the job makes progress, the estimator measures task runtime and builds a statistic, *i.e.,* it constructs an approximate cumulative distribution function (CDF) of task times. This information is then used to compute the job duration. When scheduling a new job, the Training module assigns its minimum fair share, corresponding to the minimum number of slots (a parameter of HFSP) required by the estimator to build the CDF of task times. Execution slots are assigned according to a "fewer remaining tasks" discipline, which implies short jobs are given priority. As a final note, HFSP requires an additional parameter to decide the maximum amount of slots the top-level scheduler grants to the Training module: this is useful to avoid starvation in the job scheduler, for workloads with bursty arrivals of a large number of jobs.

Before delving into the details of runtime estimators, we stress that the allocation of resources to the Training module described above and, in more general terms, in

[2] With respect to other jobs with an unknown size.

Sec. 2.2.1 are by far more important to achieve short sojourn times than extremely accurate job size estimates, as we show in our experimental results.

Runtime estimator In the following, we describe our task size estimator: note that the estimator is designed as a pluggable module. Our proposed simple estimator could be replaced by more sophisticated estimation techniques, therefore providing more accurate predictions.

Despite the intricacies of estimating Map and Reduce task time distributions (which are handled separately), the estimator is based on simple *regression analysis* to compute the parameters of a given distribution such that a measure of error (in our case least squares error) is minimized. In practice, each job can be configured such that a reference task time distribution is used by the estimator to come up with an estimated distribution based on the parameters obtained through regression analysis. In our experiments we consider a simplified setting in which there is no skew in task time distribution, which allows building job size estimates using first order statistics.

Map phase size. As observed [13, 5], across a variety of jobs, Map task execution times are generally stable and short.[3] Now, how large the sample set size should be 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. It is outside the scope of this work to come up with a mathematical model to set the sample set size. We have empirically observed that, using different data center traces, a sample set equal to five Map tasks provide sufficiently high accuracy (cf. Sect. **??**).

Let $\mathcal{M}_i$ represent the set of tasks associated to the Map phase of job i, and $\sigma(m_{i,j})$ be the duration of a single Map task j of job i. Given a sample set, for which the duration $\tilde{\sigma}(m_{i,j})$ of each Map task is *measured* while they execute, the estimator returns an estimated CDF that characterizes the whole distribution of task times. The estimated CDF is then used to produce a vector of the form:

$$\mathcal{M}_i = [\tilde{\sigma}(m_{i,1}), \tilde{\sigma}(m_{i,2}), \cdots, \tilde{\sigma}(m_{i,j}, \cdots].$$

The Map phase duration $\theta(\mathcal{M}_i)$ is the **sum** of the duration of all Map tasks, discounted by the amount of work done by tasks scheduled in the Training module.

Reduce phase size. 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, that we label *execution time*. Since a Reduce tasks can be orders of magnitude longer than Map tasks, we aim at providing an estimate of their duration before their completion. Let $\tilde{\sigma}(r_j)$ be the estimate of the *execution time* of a Reduce task $r_{i,j}$ of job i; as a first approximation, we ignore the Shuffle and sort times, and we compute $\tilde{\sigma}(r_{i,j})$ as follows:

$$\tilde{\sigma}(r_{i,j}) = \frac{\Delta}{p_k} \; \forall j \in \mathcal{T}_i.$$

[3]The training of maps requires to pay attention to data locality problems: we try to avoid to do training with non-local tasks.

Δ is a configurable parameter (expressed in seconds) that sets the trade-off between estimation accuracy and speed, and $p_{i,j}$ is the *progress* done by task $r_{i,j}$ during the execution stage. The progress of a task is computed as the fraction of data processed by a Reduce task over the total amount of its input data. This information is available once all Map tasks are done producing the intermediate output data, which is materialized locally in each TaskTracker. As such, $p_{i,j}$ embeds the information on the skew of the distribution of Reduce task times. In other words, $\tilde{\sigma}(r_{i,j})$ is a measure of the I/O throughput of a Reduce task while reading its input data from disk, normalized by the eventual skew in input data sizes. Finally, note that Δ establishes the maximum amount of time a Reduce task will remain in execution for size estimation purposes, which constitutes a bound on the training time.

The estimator operates on a sample set $\mathcal{T}_i$, for which the duration $\tilde{\sigma}(r_{i,j})$ of each Reduce task is *measured*. Then, the estimator returns an estimated CDF that characterizes the whole distribution of Reduce task times, using the available information on the skew of Reduce task input size distribution. The estimated CDF is then used to produce a vector of the form:

$$\mathcal{R}_i = [\tilde{\sigma}(r_{i,1}), \tilde{\sigma}(r_{i,2}), \cdots, \tilde{\sigma}(r_{i,j}, \cdots].$$

The Reduce phase duration $\theta(\mathcal{R}_i)$ is the **sum** of the duration of all Reduce tasks, discounted by the amount of work done by tasks scheduled in the Training module.

2.2.3 Job Preemption

The HFSP scheduling discipline uses preemption: a new job can suspend tasks of a running job, which are then resumed when resources become available. However, traditional preemption primitives are not readily available in Hadoop. The commonly used technique to implement preemption for scheduling jobs in Hadoop is that of “killing” tasks or entire jobs. Clearly, this is not optimal, because it wastes work, including CPU and I/O. Alternatively, it is possible to Wait for a running task to complete, as done in [13]. If the runtime of the task is small, then the waiting time is limited, which makes Wait appealing. While the Wait method is easy to implement and may provide good results, there are cases – tasks with long runtime – where the delay introduced by this approach may be too high.

In this work, we study the benefits of a more traditional approach to preemption, which we call **eager preemption**: tasks or jobs can be suspended in favor of other jobs, and resumed when they are awarded resources. Eager preemption requires the implementation of Suspend and Resume primitives. In our implementation, we 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 launched 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`. When HFSP suspends a task of a job, the underlying OS eventually proceeds with its materialization on the secondary storage (in the swap partition) if and when its memory is needed by another process. 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.

As discussed Sect. 2.2.1, the **job scheduler** allocates cluster resources to jobs that finish first, as computed in the virtual cluster. A new job arriving in the system may induce – depending on its size – the job scheduler to Suspend a running job. In practice, the job scheduler suspends tasks, rather than jobs: task suspension works as follows. Upon reception of a heart-beat from a TaskTracker, the job scheduler verifies whether a job tagged for suspension occupies resources. If this is the case, it proceeds with the suspension of a task of that job. This step is repeated until all tasks of the new job obtain resources. The selection of which job, among those running in the cluster, to tag for suspension follows a simple rule: the scheduler selects for suspension the tasks of jobs sorted in decreasing order of their size, which reinforces the underlying idea of the HFSP scheduling discipline. In the following, we provide additional considerations.

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.

In practice, when the **job scheduler** decides to allocate resources to a (current) job with some (or all) of its tasks suspended, it proceeds as follows. Upon the periodic heart-beat sent by a given TaskTracker , the job scheduler verifies the presence of suspended tasks of the current job. If the TaskTracker has a free slot and hosts a suspended task for the current job, the job scheduler Resume such task. If there are no free slots, two conditions may arise: such slots are occupied by tasks of a job smaller or larger than the current one. In the former case, the job scheduler waits for such tasks to terminate; in the latter, the job scheduler Suspend tasks of larger jobs, and Resume tasks of the current job. Essentially, the Resume operation is similar to that of scheduling a new task of a job, albeit it is given higher priority with respect to the allocation of tasks of the same job on a TaskTracker occupied by a suspended task.

Finite machine resources. Suspending tasks has a cost in terms of storage space requirements. If many tasks on a single machine are suspended, context data could use a large fraction of the RAM available on a machine and eventually could also deplete the swap space. Despite this is an extreme case that arises with particular workloads (a large number of jobs arriving in decreasing size), we address it by defining a set of thresholds (with hysteresis) on the number of tasks that can be suspended. When too many tasks are suspended, HFSP switches to the Wait-based preemption technique, until conditions are met for reverting to eager preemption.

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 Experiments

We have conducted a series of experiments whereby the objective is to compare the performance achieved by HFSP with respect to that obtained with other standard schedulers, such as FIFO and FAIR schedulers. We omit the results we obtained from this deliverable for two reasons: first, the results presented in the previous deliverable (D.4.5) remain substantially unchanged; second, we produced a new set of micro-benchmarks to understand the performance of the inner components of HFSP, but these results are under submission in a double-blind conference, and will be available once we will be able to disclose them.

2.4 Conclusion

The problem of scheduling jobs in parallel systems have received a lot of attention in the past, including works that attempted at producing elegant mathematical models of such systems with the goal of studying the hardness of obtaining optimal scheduling. In this work we took a systems approach, glossing over mathematical constructs and optimality analysis: instead we were interested in studying the benefits of a size-based approach to scheduling jobs in a real system, namely Hadoop.

Our work was motivated by the realization that MapReduce has evolved to the point where shared clusters are used for a wide range of workloads, which include an increasingly large fraction of interactive data processing tasks. Existing schedulers in the state-of-the-art suggest, to overcome the inherent limitations of a simple first-come-first-served discipline, cluster resources to be shared equally among running jobs. As a consequence, we have witnessed the raise of deployment best practices in which long sojourn times were compensated by over-dimensioned Hadoop clusters. Armed with the realization that a large fraction of cluster resources were used for a small amount of time, given a selection of real-world workload traces, in this work we set off to study the benefits of a new scheduling discipline that targeted at the same time short sojourn times and fairness among jobs.

The HFSP scheduler we proposed in this work brought up several challenges. First, we came up with a general architecture to realize *practical* size-based scheduling disciplines, where job size is not assumed to be known a priori. The HFSP scheduling algorithm solved many problems related to the underlying discrete nature of cluster resources, how to keep track of jobs making progress towards their completion, and how to implement strict preemption primitives. Then, we used standard statistical tools to infer task time distributions and came up with an approach aiming at avoiding wasting cluster resources while estimating job sizes. Finally, we performed a comparative analysis of HFSP with two standard schedulers that are mostly used today in production-level Hadoop deployments, and showed that HFSP brings several benefits in terms of shorter sojourn-times, even in small, highly utilized clusters.

There are several avenues that we are considering as part of our future work. First, we will extend our experimental study to cover a wider range of workloads, including those presenting issues related to skew in task time distributions; we will

also consider the impact of failures and study in more details the implications of eager preemption from the OS perspective. Finally, we will study the problem of scheduling complex job work-flows, that result from the composition of several sub-jobs. The ultimate goal of our work is to contribute HFSP to the Hadoop ecosystem. Currently, the scheduler presented in this work is released as an open source project, and we will work toward a production-ready version of HFSP for its discussion within the Hadoop community.

3 Benefits of operating a dCDN

This section presents simulation results that demonstrate the benefits of operating a dCDN, both from the users and the operators point of view. It also describes a prototype implementation of NF-DASH.

3.1 Evaluation of Workload and Overall Cost

3.1.1 Simulation Setup

In order to assess the impact of dCDN operation both on network performance and on delivered QoS, we developped an ad-hoc simulation program (in C) that tested several configurations for a video distribution service.

We used a one-week content download trace of Orange VoD service for the period June 12, 12 o'clock, to June 19, 12 o'clock, 2011. The structure of each record in the trace is illustrated in figure 3.1. The first field identifies a user and the second identifies the video or clip that is downloaded by the user. Region indicates the user's geographical location and the last field is the time when the video was requested. Since the duration of a download session is not given in the trace, we assumed that it followed a uniform distribution from 60 minutes to 120 minutes. The average session's duration is thus 90 minutes. The trace contained 364,663 download requests for 17,223 distinct videos. These requests come from users distributed in 13 geographical areas in France, which are considered to set up the dCDN topology.

user ID	item ID	region	timestamp

Figure 3.1: Structure of trace record

We assume that each of the 13 regions contains a dServer. The dServers are identical with storage capacity of 5,000 videos and serving capacity of 500 users. The single CDN server is much more powerful than the dServers as it can store all required videos and can serve up to 5,000 users.

The dCDN operation is as follows : a user's request is directed by the dCDN to a dServer (according to one of the simulated policies specified later). If this dServer is saturated (i.e. already serves 500 requests), the request is forwarded to another server that stores a replica of the video (either another dServer or the CDN server).

If the CDN server is saturated, the request is queued in order to postpone the video delivery. Only the CDN server can queue requests.

Network's topology is shown in figure 3.2. The red circle represents the CDN server located at the Point of Presence (PoP) in Paris. Each dServer is directly connected to the CDN server. These connections represent external links (i.e. toward CDN provider or transit networks). Traffics transfered on red links will be charged (either by the CDN provider or by a transit network provider), and thus generate external costs for the ISP.

In both placement and redirection policies, we may assume that dServers are organized in cooperation groups, where dServers in a given cooperation group are geographically close. For example, the cooperation group for the dServer located in Rennes includes the dServers in Caen and Orleans, while the cooperation group for the dServer located in Nice includes the dServers in Toulouse, Montpellier, Lyon and Dijon.

Each pair of geographically adjacent dServers is connected by an internal link shown by a dashed blue line in figure 3.2. Internal links are intra-domain links managed by the ISP, therefore cheaper to use than external links. Specifically, we assume that the cost for transferring a unit of data on a blue link is a half of the cost for transferring a unit of data on a red one.

(1.5,1.6)

3.1.2 Measurements

In order to highlight the benefits yielded by NF-DASH, we assess the impact of both placement and redirection policies on the cost and performance of the video retrieval service, by comparing it to a benchmark centralized solution where all requests are directly treated by CDN server. The considered NF-DASH configurations are the following combinations of placement and redirection settings:

- Random (video) placement, where a video is replicated on randomly selected dServers.

- Optimal placement, where each video follows an optimal placement strategy described in D4.4.

- Random (request) redirection, where each request is initially randomly assigned to a dServer. If either the dServer does not store the video, or if it is saturated, the request is redirected to the CDN server.

- Optimal redirection, where a user's request is first assigned to its closest dServer. If the request cannot be satisfied by this dServer, it will be redirected successively to the dServers in the cooperative group of the initially selected dServer. If none can deliver the requested video, the request is finally redirected to the CDN server.

We thus consider four implementations of NF-DASH differing on the placement and redirection policies: Rand-Rand and Opt-Rand respectively represent random

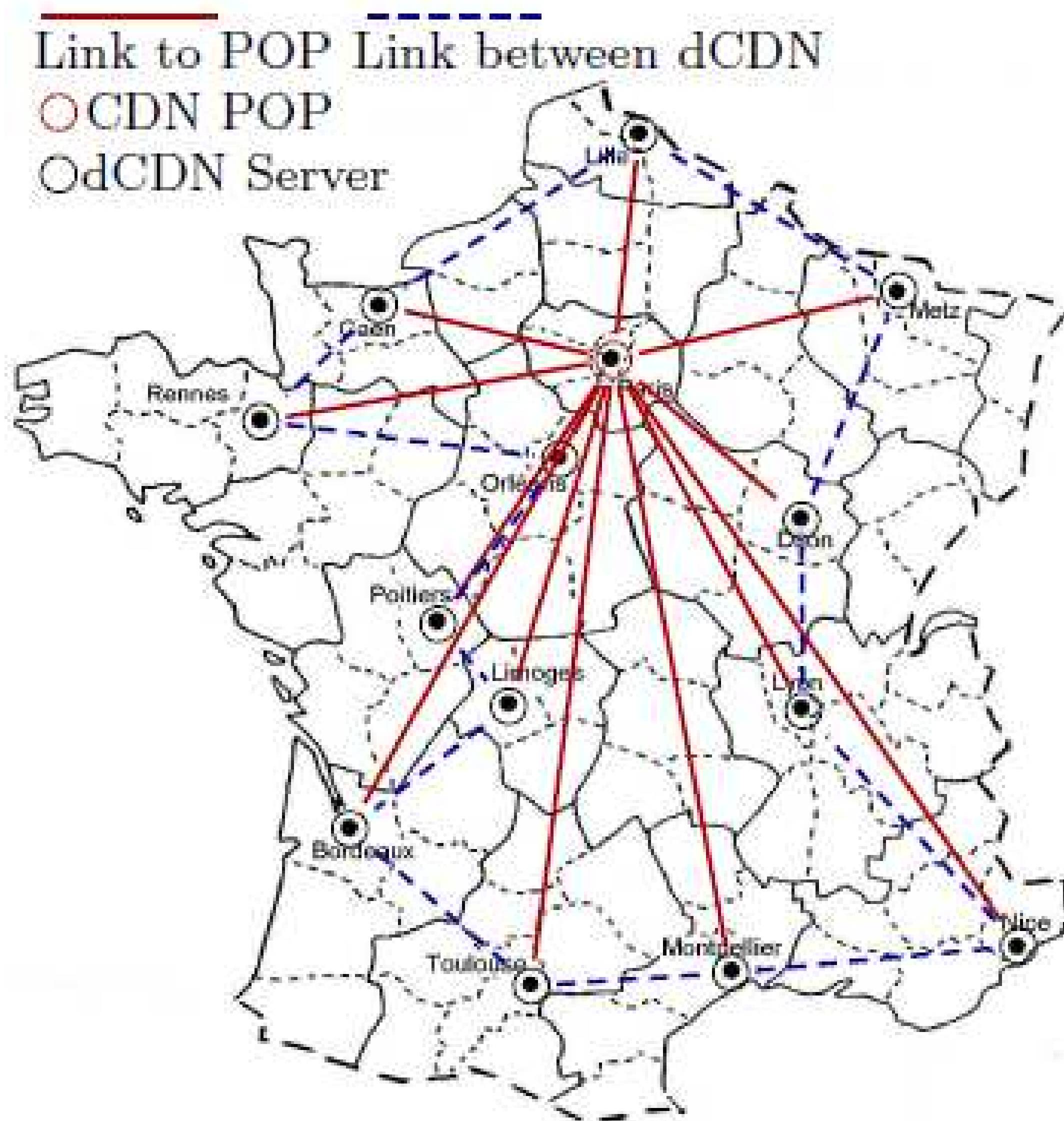

Figure 3.2: Simulated dCDN topology; the cost of using an external (red line) link is twice the cost of using a peering (blue dashed line) link.

and optimal placement with random redirection whereas Rand-Opt and Opt-Opt respectively represent random and optimal placement with optimal redirection. Note that the CDN is responsible neither for placing the videos in the dServers nor for redirecting the requests to appropriate dServers.

The metric used to assess the cost of serving a single video is the geographical distance between servers multiplied by the bandwidth consumed by video streaming. Since we assume here that all videos have the same playback bit-rate, the cost is proportional to the distance between the user and its server. The performance delivered to users (QoE) is assessed by the delay observed in serving requests (neglecting redirection times). We evaluate the number of requests that are postponed due to the saturation of the CDN server, and the average response time to postponed

requests. The performance of the considered policies is also assessed by comparing the numbers of requests served by the dServers and by the CDN server.

3.1.3 Simulation Results

All simulation results are obtained by running each simulation 100 times, and then averaging the 100 results.

First, we show the ISP operation cost produced by different configurations in figure 3.3 which represents each cost as a fraction of the cost for the centralized CDN scheme.

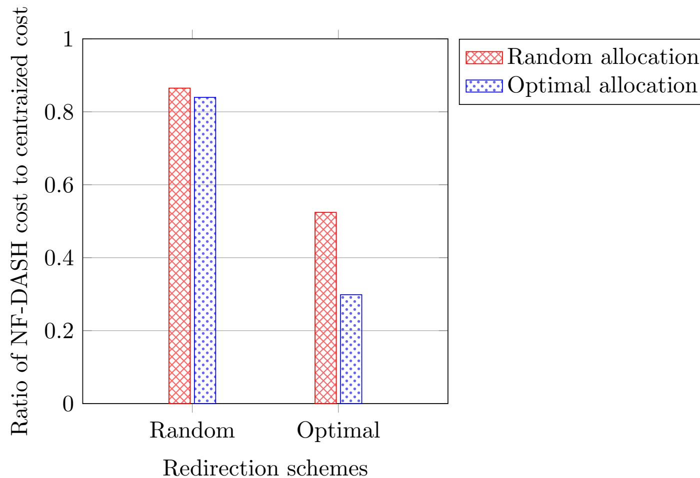

Figure 3.3: Comparison of ISP operation costs

Figure 3.3 shows that all configurations clearly outperform the centralized solution. Even a simplistic NF-DASH implementation where contents are randomly placed in dServers, and requests are randomly directed to dServers (the Rand-Rand configuration) allows to reduce the cost by about 15%. The redirection strategy is shown to have the major influence on operation cost: for random redirection, otimizing placement (the Opt-Rand configuration) presents a limited gain compared to random placement (the Rand-Rand configuration). On the other hand, in case of optimal redirection, the operation cost is drastically lowered. In the random placement case (the Rand-Opt configuration), the cost is roughly half of the cost of the centralized CDN, whereas optimizing also the placement (the Opt-Opt configuration) allows to reduce the operation cost to only 30% of benchmark's operation cost.

In figure 3.4, 3.5 and 3.6, each curve represents one configuration of NF-DASH or the centralized solution.

Figure 3.4 demonstrates the overall workload undertaken by the dCDN system. Keep in mind that the task of the dCDN is to quickly reply to video requests in order to avoid delay. So a large workload on the dServers translates into a lower workload on the CDN server and less postponed requests (i.e. improved QoE). First of all, we see that the workload's variance shows clearly a daily periodicity. Two daily

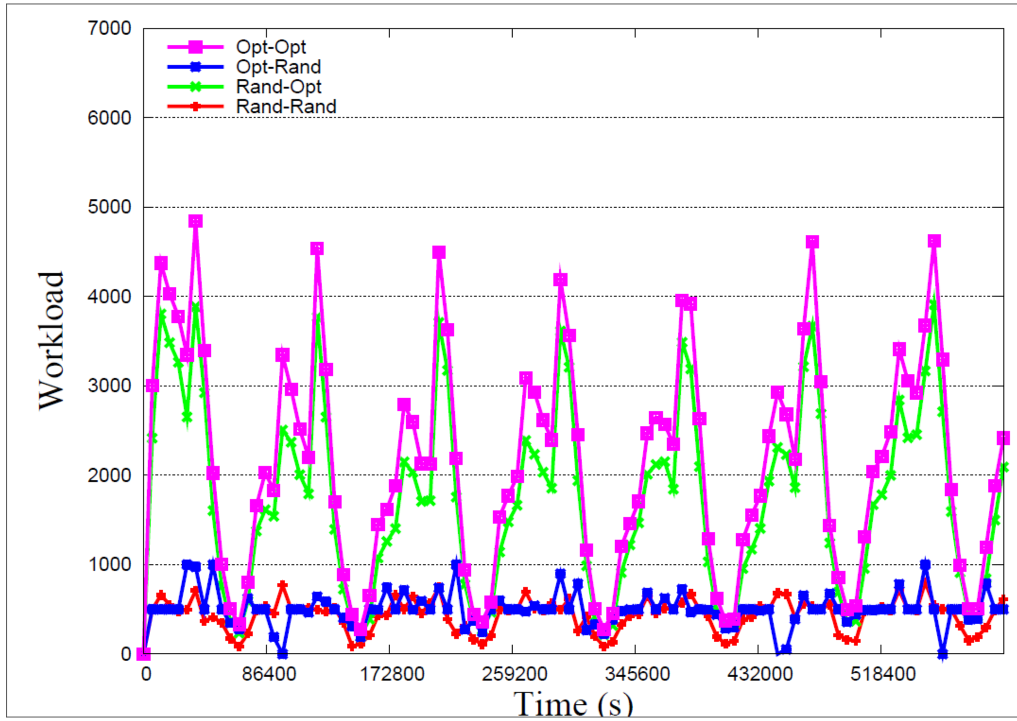

Figure 3.4: Workload on dServers

peaks happen at around 2:00 P.M. and 11:00 P.M, and the workload is minimum at around 4:00 A.M. As the trace starts at Sunday noon, we observe the difference of the workload on week days and weekends. The peak of the workload decreases from Sunday to Thursday, and then increases again on Friday and Saturday nights. These characteristics matches perfectly the user behavior in real world. Thus, the results obtained from our trace are valuables references for future systems designs. Obviously, the Opt-Opt configuration treats the largest number of requests. At the "rush hour", the number of requests served by dCDN servers is close to 5,000. This number indicates that almost all service capacity offered by the NF-DASH is occupied. The Opt-Opt configuration outperforms the Rand-Opt by about 25%, but both perform well. On the contrary, the performances of Rand-Rand and Opt-Rand configurations are not ideal. The utilization ratio of dCDN bandwidth is only 10% even in the peak period. This proves once again that the redirection scheme is a critical issue concerning the dCDN performance.

Corresponding to figure 3.4, figure 3.5 gives the workload dealt with by the CDN server in various NF-DASH configurations and the centralized CDN configuration. The advantage of Opt-Opt configuration stands out prominently in the figure. When Opt-Opt is implemented, the peak workload at 2:00 P.M. almost disappears at the CDN side. During the night peak on weekdays, only around 70% of the CDN bandwidth is occupied in Opt-Opt, while the download link is always saturated in the centralized solution. At weekends, the heaviest workload in all configurations reaches the bandwidth capacity of the CDN server. However, the Opt-Opt configuration yields the shortest duration of the saturated state. The Rand-Opt configuration also alleviate the CDN workload visibly, and the reduction is about 10%. The other two

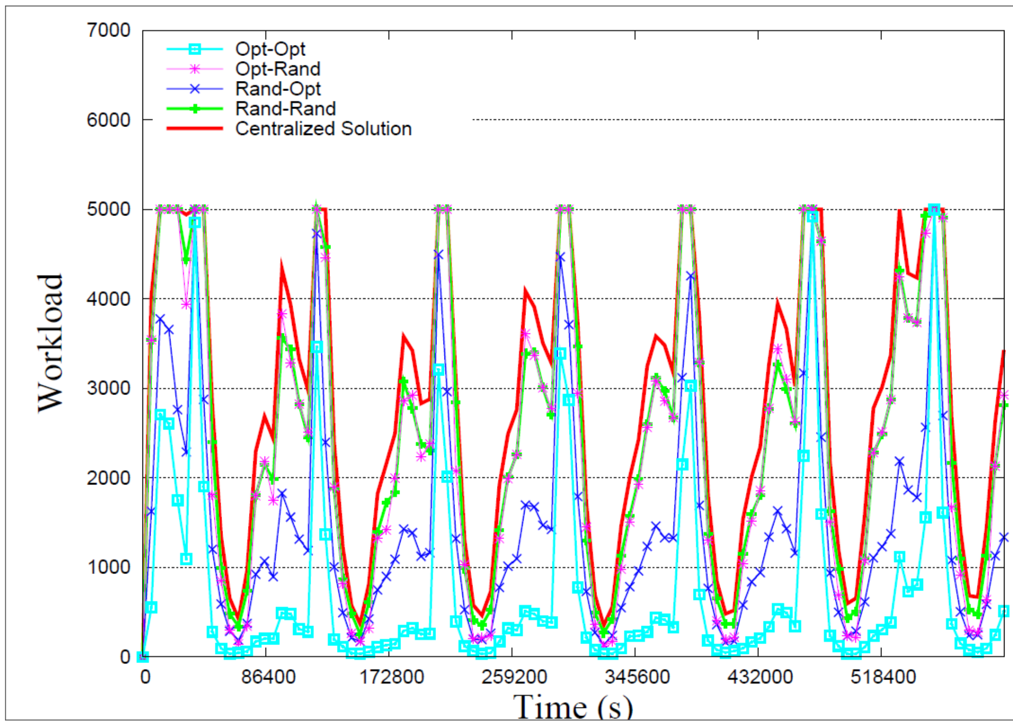

Figure 3.5: Workload on CDN server

Measurements / configurations	Access to dCDN	Access to CDN	Average delay
Centralized	0	100%	4639s
Rand-Rand	13.5%	86.5%	1716s
Rand-Opt	39.2%	60.8%	1220s
Opt-Rand	16.1%	83.9%	1705s
Opt-Opt	76.9%	23.1%	886s

Table 3.1: Distribution of access and Average waiting time

configurations only slightly diminish the bandwidth utilization of the CDN server.

Figure 3.6 counts the number of requests postponed by the CDN server. Same as it is shown in previous figures, the Opt-Rand and Rand-Rand configurations do not lessen much the number comparing with the centralized solution. The Rand-Opt configuration clears up all delayed requests during week days. However, the number at weekends still exceeds 1,000, which is not acceptable for VoD service. Fortunately, the Opt-Opt configuration performs satisfactorily. There is almost no delayed request all the week, except at Saturday night. And the number is less than 100.

In table 3.1, we show the percentage of requests that are respectively satisfied by dServers and by the CDN server, together with the average waiting time of delayed requests. All NF-DASH configurations largely reduce the waiting time. Among all configurations, Opt-Opt works extremely well : close to 80% of the requests are served by the dCDN and the waiting time for postponed requests is dramatically shortened.

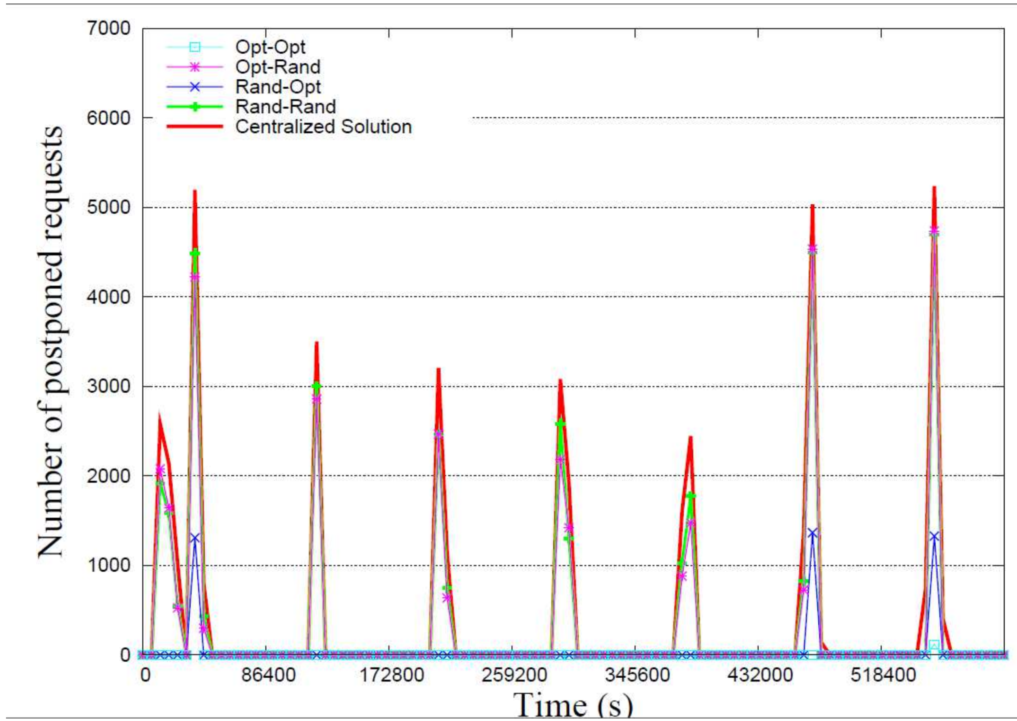

Figure 3.6: Delayed requests at CDN server

3.1.4 Prototyping NF-DASH

The objective of designing a prototype is to show the feasibility of the NF-DASH architecture and to measure some basic performance metrics like the gain in inter-operator links load and the enhancement in the quality of experience of clients.

The implementation of a dCDN must be as simple as possible and must be compatible with existing technologies. We propose to use the current specification and the current implementation of DASH VLC clients and to push the intelligence of the system into the operator's network. We considered videos structured into equal duration chunks, and coded into several formats. In order to distribute the work between different servers, we partitioned the network into geographic areas which contain their own sets of dServers.

When connecting to a classical CDN server, a client requests an MPD file containing URLs of the video chunks. In our prototype, the original CDN server redirects the client to an *MPD generator server* located in the dTracker of the client's operator network. An MPD generator Servlet creates an MPD file which is client specific. The URLs provided in this file are not directly leading to chunks on dServers, as they are requests to a redirection Servlet located in the client's geographic area ; this allows managing sets of dServers per geographical area. Receiving such a request, the redirection Servlet can in turn redirect the client to the appropriate (e.g. the nearest available server) dServer hosting the requested chunk. In this way, the operator can decide from where the client will retrieve the chunks and thus optimize chunks' location depending on dServers and network conditions.

To test the deployment of our prototype architecture, we have implemented all

the components (dTracker and dServers) and deployed them on a VIPEER inter-parnter platform. Each partner hosts a dServer and several streaming clients. A single dTracker is located in TELECOM Bretagne, Brest.

3.2 Evaluation of Impact on Infrastructure

3.2.0.1 Network topology and management

As for the management of this network, we consider a policy, which is today representative of the management decision that have to be taken by an ISP. There exist three categories of links. The costs are indicated in a unit of money (euros in our case) per kilometer and per transmitted video.

The **peering links** connect each repository to the PoP. The traffic on this link goes through the PoP, so it generates peering costs as well as possible monetary compensations to the CDN. We suppose the cost of peering link is 1,000. These are red plain lines in Figure 3.2.

The **internal links** connect a pair of geographically adjacent repositories in the backbone. They are dashed blue lines in Figure 3.2. These links are intra-domain links, which are much cheaper than peering links. Therefore, the cost of internal link is set to 1.

The **low-priority internal links** are regular internal links but the network operator experience troubles on these links (they are over-used, or subject to faults). In our simulations, we chose three links: from Lille to Metz, from Lyon to Nice and from Limoges to Poitiers. Since the network operator prefers to use them in low priority, we set a cost of 1,000 but these links are shorter than peering links, so they still represent an opportunity to avoid going to the PoP.

The telco-CDN system operates as follows: an end-user's request is firstly directed to its regional repository. If the regional repository does not have the required video or its bandwidth capacity is over, this repository explores its cooperation group. A cooperation group is defined for each repository j as a set of repositories whose distance to j is smaller than the distance from j to the PoP. Any repository in the group having the requested video and free bandwidth can serve the client. If the requested video is stored within the cooperation group, the request is redirected to the matching repository. Otherwise, it is a miss and the request is forwarded to the PoP.

3.2.0.2 Evaluated Strategies

Random placement extracts the videos requested during the warm-up. Then each unit of storage space is randomly filled with one video.

Proportional placement distributes the replicas of each video according to its popularity. The number of replicas is proportional to its request rate during the warm-up, and each repository holds only one replica of a video.

Push strategies result from our PGA. To investigate the impact of the recommendation accuracy, we produce the prediction of users' preference by mixing up warm-up and test part. Specifically, we replace a certain percentage of records in the

test part with records from the warm-up part to generate predictions with various qualities. In particular:

- A **Perfect Optimal** (or Perf-Opt) takes in input of the computation for he k-PCFLP all the requests that will actually be requested during the test period. It is like the recommendation system was prophetic.
- A **Realistic Optimal** (or Real-Opt) takes in input of the computation only requests from the warm-up period. The recommendation engine does not predict anything but just rely on the past.

Perf-Opt and Real-Opt can be regarded as the upper and lower bound of our optimal push strategy.

LRU caching strategy implements the traditional and widely adopted caching strategy. To avoid unfairness due to initially empty caches, we measure from the first request of the test period. When a request cannot be fulfilled at a cache, the repository looks for possible hit in its cooperation group. However, in case of miss, the video is not re-forwarded to these repositories.

3.2.1 Results

We enter now in the evaluations of the strategies in our toy-telco-CDN. The main criteria that matters in our study is the overall cost. For each placement strategy, we compare the cost to the one without repositories, that is, we compute the normalized overall cost as the ratio of the overall cost from fetching the video to the cost using only peering links.

3.2.1.1 Computation time of quasi-optimal tracker

We do not aim here to analyze the performances of our genetic algorithm, and to explore the benefits of our parallelized implementation. However, we can report some basic computation time to express that the implementation of a quasi-optimal tracker is possible using today's technologies.

The algorithm has been implemented on a MR cluster consisting of 10 machines with dual 2.70 GHz Pentium processor and 4 GB RAM. We set the population of each generation to 500 individuals and the initial population is stored in files whose overall size is about 150 MB. For all instances, the algorithm converges in about 350 generations, which takes less than 12 hours. With regard to the relatively sub-optimality of our implementation and to the small number of computers involved in this computation, it is clear that the implementation of a quasi-optimal tracker can easily be implemented at a large-scale, typically in the context of such service where placement is re-done on a daily basis.

3.2.1.2 Impact of the recommendation accuracy

Our tool enables the comparison of various parameters on the overall network performances in a fair manner. Here we explore as an example the importance of the accuracy of the recommendation engine.

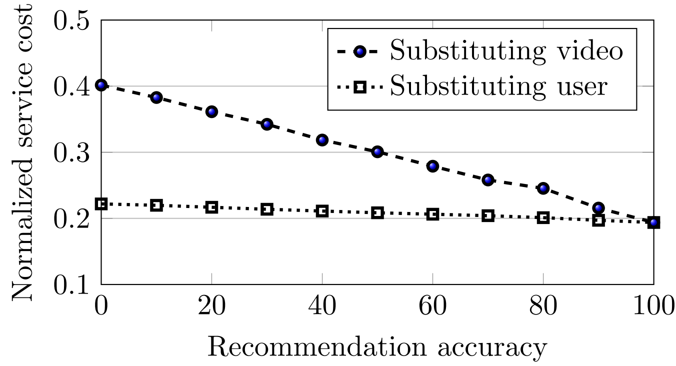

Figure 3.7: Impact of prediction on Perf-Opt

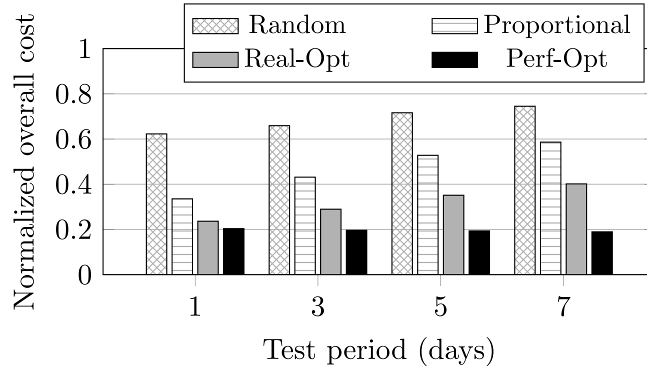

Figure 3.8: Normalized overall cost of push strategies

The recommendation engine provides a set of p_{ik} that reflects that end-user i will request video k in the near future. We did not implement any recommendation engine, but we emulated a recommendation engine with various levels of performance. In the set of non-null p_{ik} that are given in input of the algorithm, a percent of the requests have been replaced by requests from the warm-up part. The perfect recommendation engine (accuracy $a = 100\%$) is thus able to predict all requests. The realistic recommendation engine (accuracy $a = 0\%$) does not predict any request.

When we replace a recommendation, we can either substitute either the requested video or the requester. We show in Figure 3.7 the performances for a series of accuracies.

Substituting videos gives a larger bad effect on the efficiency of our optimal placement. The influence of the mistakes on users may be canceled by the integration of users' requests at the regional repository. Therefore, only marginal increment of cost is yielded. However, the error of pushing wrong videos is not compensable, which doubles the overall cost. Thus, we emphasize here the importance of addressing the videos that will be popular in the near future at a regional scope rather than at individual ones.

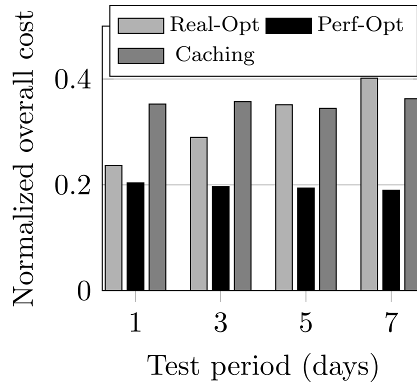

Figure 3.9: Overall cost of Optimal and LRU

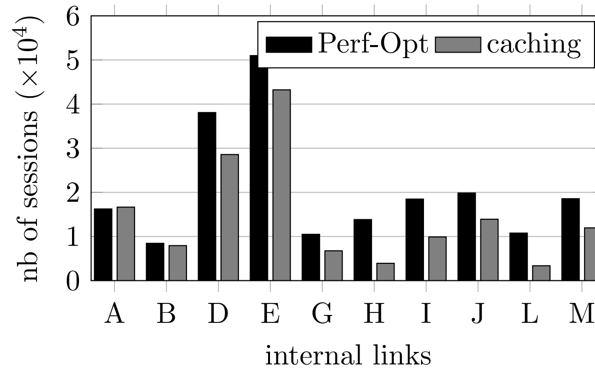

Figure 3.10: Internal link utilization

3.2.1.3 Performances of push strategies

We study the push strategies, especially both random and proportional to the quasi-optimal strategies. Figure 3.8 shows the prominent performances of our quasi-optimal computation. Although the proportional random strategy is widely accepted as a decent heuristic for placement in terms of hit ratio, we show here that the performances are bader than the optimal placement in terms of telco-CDN management.

When the test period is longer, the performances degrades as some videos that were not requested during the warm-up enter in the catalog, thus are requested. Obviously, the Perf-Opt strategy is not affected by such novel videos, but we observe a significant degradation of the performances of Real-Opt (almost twice less efficient). We mitigate this observation by arguing that placing videos on a weekly basis is not serious when it is possible to do it on a daily basis.

3.2.1.4 Push strategies versus caching

What follows is the main outcome of this paper: we compare, on a given network configuration and a given VoD service, the advantages of a push strategy versus a caching one. The overall performances of LRU caching in term of cost is com-

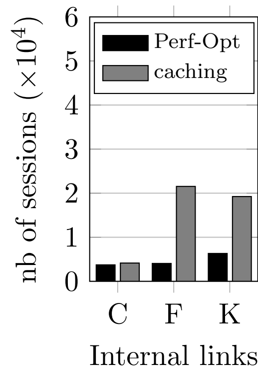

Figure 3.11: Low priority link utilization

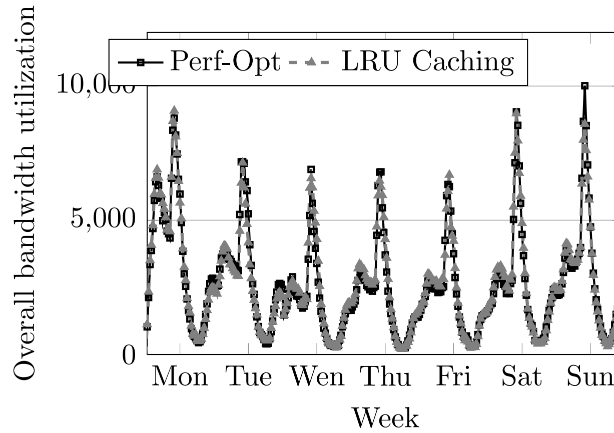

Figure 3.12: telco-CDN bandwidth utilization

pared with push strategies in Figure 3.9. Please note that the performances of both caching and Perf-Opt are not affected by the duration time, the former because it is prophetic, the latter because it dynamically uploads the content in the repositories. On the contrary, performances of Real-Opt degrades when no new computation is done.

For the test period of one day, which is the most probable to implement, the performances of push strategies are much better than caching. They achieve an overall cost that is 1.8 times smaller than LRU caching. This result demonstrates that LRU caching is a blind strategy that performs remarkably well for hit-ratio, with almost no implementation cost. However caching is far from ideal for an ISP that wants to actually manage its network as well as its telco-CDN.

The reason of the predominance of Perf-Opt is revealed in Figure 3.10 and Figure 3.11. Both figures count the total number of sessions passed through each telco-CDN internal link. In Figure 3.10, it is visible that a telco-CDN implementing LRU caching uses less internal links than implementing Perf-Opt. In the meantime, Figure 3.11, Perf-Opt utilizes in low priority the low-priority links, which is exactly the desire of the ISP, although LRU caching utilizes these links as regular links (note that they are still cheaper than peering links). In other words, while Perf-Opt al-

lows to engineer the traffic (here to avoid using low priority links but more accurate policies can obviously be implemented), a cooperative LRU caching acts naively.

We represent in Figure 3.12 the overall bandwidth utilization on telco-CDN repositories. As could be expected, LRU caching performs similarly to Perf-Opt. A naive scientist looking only at such results would definitely adopt LRU caching since it is far easier to maintain, but the results given in Figure 3.12 emphasizes the opposite. It is remarkable that Perf-Opt performs as well as LRU caching although its impact on the network infrastructure is 1.8 times less severe.

4 Conclusion

This deliverable has addressed three complementary issues. The first one described the design of a new scheduling algorithm for a shared cluster dedicated to MapReduce jobs. MapReduce has indeed been considered within WP4 as a method for handling computations that involve very large amounts of data, as is necessary when predicting future content demands, and when optimizing content replication and placement. The second part is dedicated to the efficiency of various prediction methods that are compared in a potential VIPEER implementation. It is shown that, although a simple frequency popularity is very efficient, the use of personal recommendation allows to identify less popular content, that are still requested. The last part has shown how a genetic algorithm can be used to implement optimal placement of the content within the ISP's domain; this method has been shown to be significantly more efficient that a simple caching strategy.

Bibliography

[1] Hadoop Mumak. `https://issues.apache.org/jira/browse/MAPREDUCE-728`.

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

[3] H. Chang et al. Scheduling in MapReduce-like Systems for Fast Completion Time. In Proc. of IEEE INFOCOM, 2011.

[4] 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.

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

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

[7] E. Friedman and S. Henderson. Fairness and efficiency in web server protocols. In Proc. of ACM SIGMETRICS, 2003.

[8] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resources types. In Proc. of USENIX NSDI, 2011.

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

[10] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: A comparative workload analysis from three research clusters. In Technical Report, CMU-PDL-12-106, 2012.

[11] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In Proc. of JSSPP, 2010.

[12] 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 ACM MIDDLEWARE, 2010.

[13] 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.