mPlane

an Intelligent Measurement Plane for Future Network and Application Management

ICT FP7-318627

Algorithm and Scheduler Design and Implementation

Author(s):	POLITO	D. Apiletti, E. Baralis, A. Finamore, L. Grimaudo, S. Traverso
	FUB	V. Guchev, F. Matera, E. Tego
	ALBLF	Z. Ben-Houidi
	EURECOM	P. Michiardi, M. Milanesio
	ENST	Y. Gong, D. Rossi
	NEC	M. Dusi (ed.), S. Niccolini, S. Nikitaki
	TID	G. Dimopoulos, I. Leontiadis
	NETvisor	Á. Bakay, T. Szemethy
	FTW	A. Bär, P. Casas, A. D'Alconzo, P. Fiadino

Document Number:	D3.3
Revision:	1.0
Revision Date:	31 Aug 2014
Deliverable Type:	RTD
Due Date of Delivery:	31 Aug 2014
Actual Date of Delivery:	31 Aug 2014
Nature of the Deliverable:	(S)oftware
Dissemination Level:	Public

Abstract:

This deliverable describes the data-analysis algorithms implemented at the repository for each use case. Furthermore, it contains the algorithms and tools to support the parallel computation of the algorithms by using a distributed cluster of resources. These tools are publicly released by the mPlane Consortium, and include:

- Hadoop Fair Sojourn Protocol, a scheduler for Apache Hadoop;
- Schedule, a tool for cache-oblivious scheduling of shared workloads;
- RepoSim, a ns2 based simulator to fine-tune the mPlane repository performance.

Software and instruction on how to access and use them at http://www.ict-mplane.eu/public/software. To highlight the strong collaboration with other EU-projects, the Hadoop Fair Sojourn Protocol has been developed jointly with partners from the BigFoot project.

Keywords: mPlane software, repository tools, job scheduler, algorithm design

SEVENTH FRAMEWORK PROGRAMME

Disclaimer

The information, documentation and figures available in this deliverable are written by the mPlane Consortium partners under EC co-financing (project FP7-ICT-318627) and does not necessarily reflect the view of the European Commission.
The information in this document is provided ``as is'', and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

Contents

Document change record

Version	Date	Editor(s)	Description
0.1	11 Aug 2014	M. Dusi (NEC)	initial draft
0.9	28 Aug 2014	M. Dusi (NEC)	final draft for review
1.0	31 Aug 2014	M. Dusi (NEC)	final

1 Introduction

This document details the algorithms that operate on the data storage layer, that are being developed since the first year after their definition, as detailed in Deliverable D3.1. As most of the algorithms are use-case specific, we opted for describing them in the context of each use case. Algorithms that have a more general nature other than the one detailed for the specific use-case, and can be exploited to support the work of the entire repository layer are also presented.

This document also describes some features of the computing environment required by the repository. In particular, we report on the design and preliminary implementation of the job scheduler *Hadoop Fair Sojourn Protocol*, for executing concurrent tasks on a shared computing environment, and of *schedule*, a tool for cache-oblivious scheduling of shared workloads. A software release of both schedulers is provided. Moreover, we provide a performance comparison of several stream-computing platforms which can potentially work within the repository to operate on the very large amounts of data coming from the monitoring probes. Finally, we describe *repoSim*, a ns2 based simulator to fine-tune the mPlane repository performance.

Together with this deliverable, we release the following tools:

- **Hadoop Fair Sojourn Protocol**, a scheduler for Apache Hadoop;
- **Schedule**, a tool for cache-oblivious scheduling of shared workloads;
- **RepoSim**, a ns2 based simulator to fine-tune the mPlane repository performance.

The code and description of such tools can be accessed from the project website at `http://www.ict-mplane.eu/public/software`.

To highlight the strong collaboration with other EU-projects, the Hadoop Fair Sojourn Protocol has been developed jointly with partners from the BigFoot project.

This document is organized as follow. First, a brief introduction is made in Chapter 1. In Chapter 2, we discuss the design of the algorithms that operate at the repository layer, with their results. In Chapter 3, we describe the design and implementation of the tools we release, as well as the performance evaluation of different stream-computing platforms. Finally, Chapter 4 concludes the document and describes the directions of future work.

2 Large-scale data analysis algorithms

2.1 Supporting DaaS troubleshooting

To detect the Quality of Experience of users accessing content using Desktop-as-a-Service (DaaS) solutions through thin-client connections, we consider statistical classification techniques to infer on-the-fly the application that runs inside the thin-client protocols, by passively observing features of packets of the thin-client connection. Given that information, we combine them with the actual network conditions along the path, such as the Round Trip Time (RTT) and the available bandwidth of the connection. A threshold-based algorithm is then able to infer users' QoE.

2.1.1 Algorithm design and description

As for the design of the statistical application identification, we evaluated the accuracy of four supervised statistical classification techniques widely-used in the traffic classification field, namely Support Vector Machines (SVM), Random Forest (RF), Naive Bayes (NB) and Decision Tree (C4.5), in detecting the applications running on top of the thin-client connections. In our evaluation we exploited the WEKA Java library 2, which includes implementations of all the aforementioned supervised machine learning algorithms.

We applied the above techniques to our dataset. In particular, we used a dataset (*D1*) to train the techniques and evaluated their accuracy on another dataset (*D2*), both in terms of bytes and number of epochs that are correctly classified. A complete description of the datasets and of the testbed is provided in deliverable D5.1 and in deliverable D4.2.

In this context, we define accuracy as the number of epochs (bytes) correctly identified by the techniques as belonging to a given application out of the total number of epochs (bytes) that belong to that specific application.

2.1.2 Results

Starting from the results described in deliverable D4.2, where we investigated how the techniques perform when the dataset used for training and the one used for testing include the same class of applications and are collected under the same network conditions, we extended the analysis to investigate the robustness of the techniques when the network conditions between the traces used for training and the ones used for testing differ. As the algorithm based on SVM showed to perform better than the other machine-learning algorithms that we considered, here we report the results only when using SVM as algorithm of the statistical application identification module.

In particular, we evaluated how the techniques perform when the thin-client (RDP) sessions under test experiment bandwidth reduction (down to 1.5Mbps on the direction from the server to the client and 256Kbps in the opposite direction), packet delay (up to 160ms) and packet loss (up to 3%).

Figure 2.1 reports the accuracy results for each category of traffic when training the SVM technique on the dataset *D1* and testing it on traces of the dataset *D2* on which we changed the bandwidth (i.e., traces with neither packet loss nor delay). The figure outlines the results when considering a

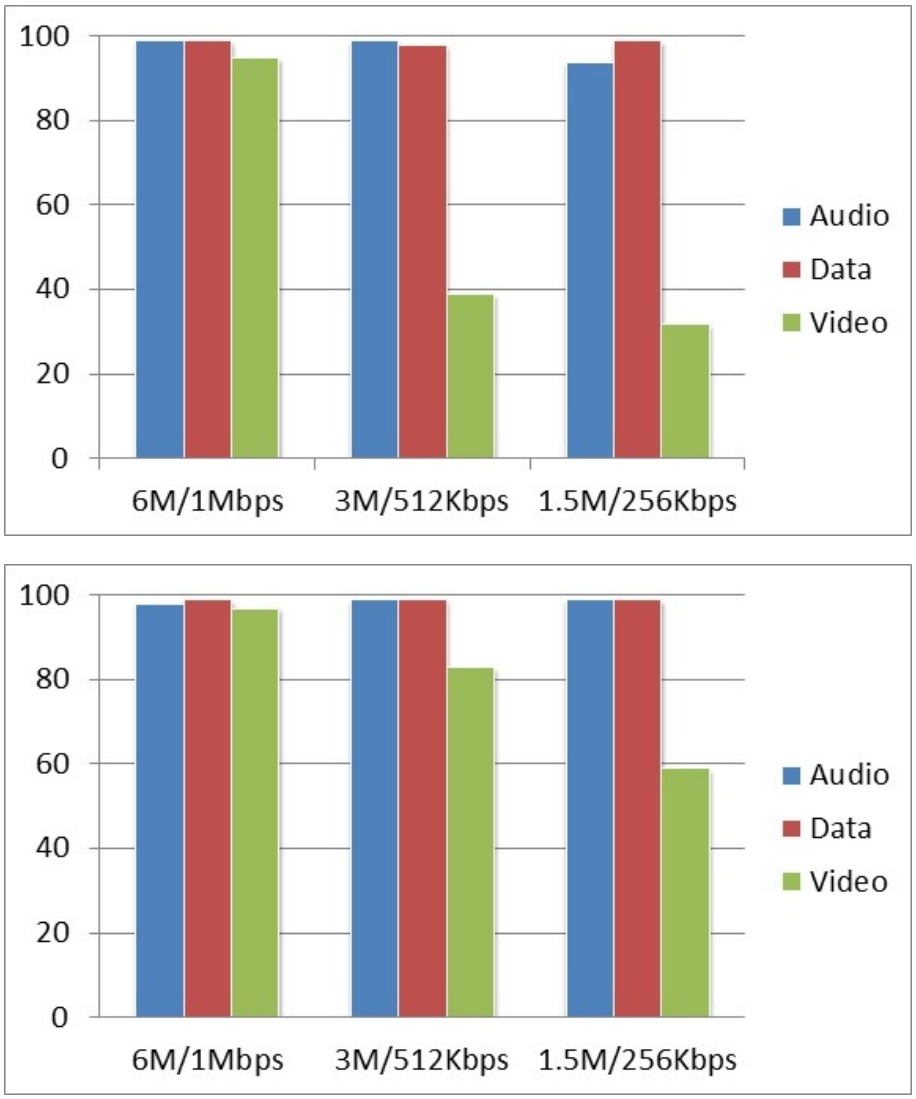


Figure 2.1: Robustness of SVM to network conditions: percentage of accuracy per epochs [top] and per bytes [bottom] on a ten-second time window. Note that the training has been done without considering any network impairments.

time-window of ten seconds (note that the algorithm did not receive any training for the network conditions under which it was tested). While the categories *Audio* and *Data* are not affected by the bandwidth reduction that we applied, the accuracy of the category *Video* decreases, as we reduce the bandwidth, from 95% to 32% in terms of epochs and from 97% to 59% in terms of bytes. As the numerical results suggests, the epochs that are misclassified are the ones that carry the minority of bytes. Moreover, we evaluated how parameters such as packet loss and packet delay affects the accuracy results. On our data, the effect of packet delay turns out to be negligible: a delay up to 160 milliseconds has no influence on algorithms that take decision based on observation windows of the order of seconds. On the contrary, packet loss has an effect on the results: when using SVM on a ten-second window, the accuracy goes down to 67% (86%) in terms of epochs (bytes) with traces that have 3% packet loss.

2.2 Estimating content and service popularity for network optimization

To detect web contents that show the potential of attracting large future popularity, we designed algorithms to (i) extract popular contents from the HTTP log repository online, (ii) train and generate the signatures of request arrival processes using a Hierarchical Clustering Structure, and (iii) identify popularity patterns through a likelihood maximization algorithm.

We focused on specific web services such as YouTube and kind of files such as JPG images and MP4 videos, as an ISP may be interested in predicting the popularity of YouTube videos to proactively push them to the caches closer to the users.

2.2.1 Algorithm design and description

Probes at WP2 monitor the HTTP requests that are generated by users to download given kind of content: at the repository we exploit such information to build a time series of arrival requests for each observed content, that we employ as training set.

We describe the behavior of individual content over time by means of Gaussian Mixture Models (GMM): GMM are known to provide greater flexibility and precision in modelling the underlying statistics of the sampling data. Given an object o_i, we account for the curve of its observation data set $\mathcal{D}(o_i)$ given by the number of downloads for a specific time interval. The curve is fitted by the linear combination of K Gaussian distributions, *i.e.* a GMM. In order to overcome the computational load implied by traditional mixture estimation techniques, during the mixture fitting we adopt a new algorithm denoted as Factorize Asymptotic Bayesian (FAB) [29]. In particular, FAB algorithm outperforms state-of-the-art Variational Bayesian methods for the fitting of the considered data set in terms of both model selection performance and computational efficiency.

After modeling the request pattern of each of the observed objects, an agglomerative hierarchical clustering algorithm is adopted in order to build the cluster tree. In order to measure the similarity of two objects the Jensen-Shannon (JS) divergence is adopted as dissimilarity metric. At the first step of the algorithm each cluster is composed by a single object o_i. Then at each step, the method finds the pair of clusters to merge in a single parent cluster, such that to minimize the increase of the total within cluster variance after merging. To merge the two data sets $\mathcal{D}(o_i)$ and $\mathcal{D}(o_j)$, the direct approach considers the new data set $\mathcal{D}(o_m) = \mathcal{D}(o_i) \cup \mathcal{D}(o_j)$. Then, the model parameter

estimation algorithm is applied to $\mathcal{D}(o_m)$ and the GMM model parameters of the parent data set, θ_m, are obtained. This completes the training stage.

To predict the popularity of an unknown content, we run the following algorithm online: whenever a content is observed for r requests, we update the GMM signatures modeling its popularity evolution and match them against the GMMs in the hierarchical clustering structure. To identify the best pattern class we run a Likelihood Maximization algorithm: eventually, it returns a request arrival process which we employ to predict the future popularity of observed content.

2.2.2 Results

We run our prototype on a set of anonymized traces from a commercial ISP, which collects the requests of YouTube videos from a population of 28000 users. First, we gathered the models of our target applications by collecting a set of requests for 10000 YouTube videos over time (aggregated by day) that served as training set. Then, we tested the validity of such models on a subset of 3400 requests to videos available in our trace.

We started investigating the accuracy of our algorithm in predicting the future popularity of the videos provided that the algorithm has observed the evolution of the requests of the videos in their first x days. Figure 2.2 reports the results of the mean percentage error when the history samples are equal to 31 and 91, respectively. The mean percentage error (MPE) is defined as the ratio between the absolute estimation (the difference between the estimated and the real requests) and the number of requests that the video actually gets in the data sample during the prediction. The performance of the algorithm is influenced by the number of data samples of the video requests. As it was expected, the accuracy of the algorithm decreases for long future horizons. However, it is interestingly observed that the median and 25% percentile error of the algorithm remain relatively constant when predicting in steps further into the future for both history samples, indicating that the performance is affected by a small number of outliers.

To further confirm that the performance of the algorithm is highly affected by a small number of videos, in Figure 2.3 we report the cumulative distribution function ($P\|X\| \leq x$) of the percentage error for various prediction steps for the history samples 31 and 91. Specifically, for 91 sample history and in the extreme case of predicting in 9 future steps only 20% of the objects highly influence the performance of the algorithm. The pattern request of the specific videos is characterized of the abrupt increase of user's request, a behavior that GMM finds hard to capture.

2.3 Passive content curation

To implement a crowd-sourced service to assist users in identifying relevant content on the web, we designed a large scale data analysis algorithm that runs online on the HTTP log repository to extract the URLs that were clicked by users (user-URLs) as opposed to those that were automatically queried by browsers and other applications that exploit HTTP protocol.

In deliverable D4.2, we presented a set of heuristics that allow to extract user-URLs and that can work offline. In the following we describe how we modified such heuristics to be able to work in an on-line fashion.

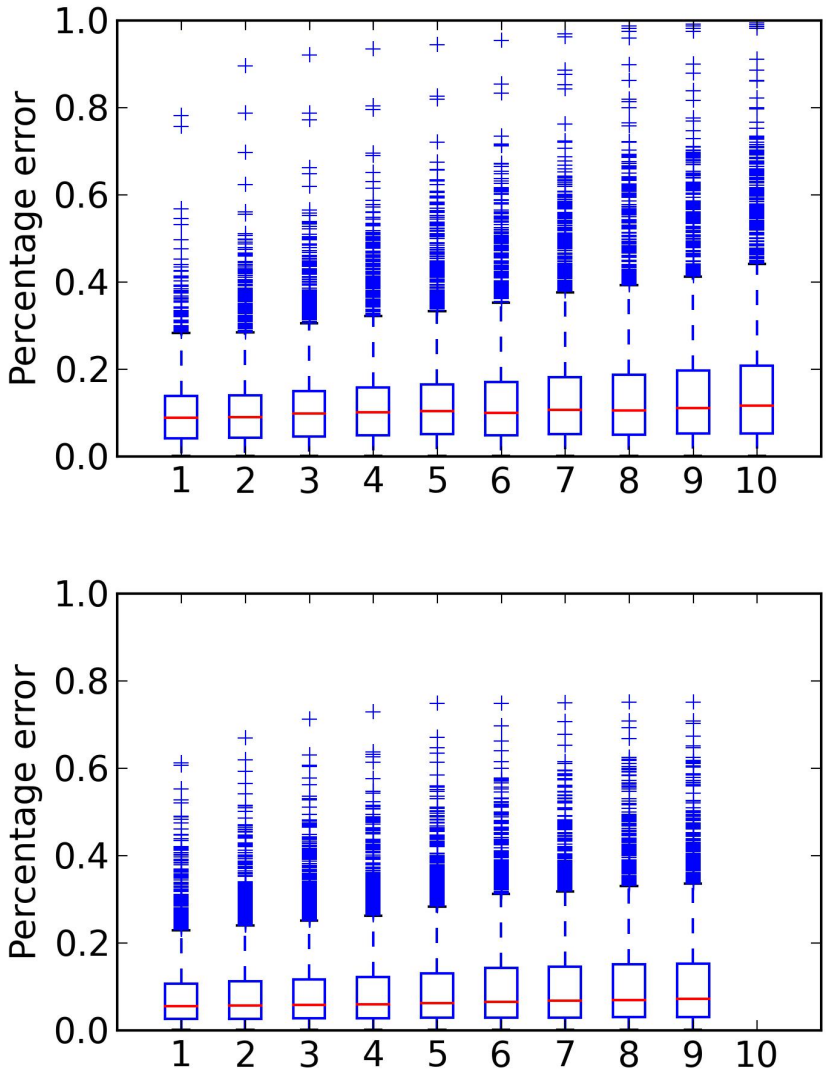


Figure 2.2: Box Plot of the mean percentage reconstruction error when predicting the popularity of the video in different future steps for history length of 31 [top] and 91 [bottom].

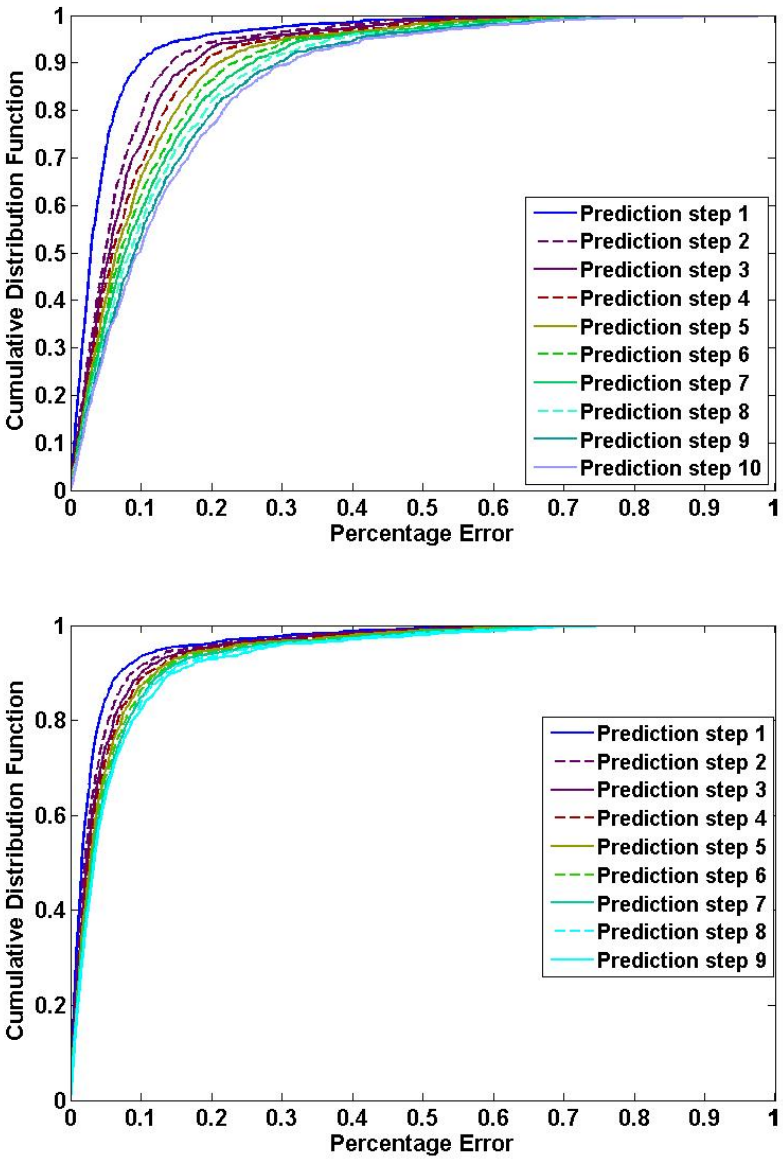


Figure 2.3: Cumulative distribution function of the percentage reconstruction error when predicting the popularity of the video in different future steps for history length of 31 [top] and 91 [bottom].

2.3.1 Algorithm design and description

During the design of online algorithms, and moving to real traffic traces (as opposed to synthetic traces), we noticed two sources of problems for which we design two simple countermeasures.

i) *URLs generated by non-browser applications:* Modern applications use HTTP to automatically download web objects, e.g., software updates. Those URLs are clearly not relevant, and must be ignored. To this extent, we leverage the *User-Agent* field exposed in HTTP requests, which informs about which client application generated the request. A white-list of well-known browsers is enough to discard roughly around 15% of URL generated by non-browser applications.

ii) *Inflated popularity induced by few users:* Web browsers can generate multiple HTTP requests for the same URL, e.g., automatically reloading a page, or downloading videos in chunks. This phenomenon inflates the popularity of some URLs. We counter this effect by ignoring user-URLs for which the Referer field is the same as the user-URL itself.

With this pre-filtering techniques in mind, we then adapted our prior heuristics to be able to run online. As described in D4.1, our heuristics are: (i) the referer-based filter (F-Ref), (ii) the Type-based filter (F-Type), (iii) the Ad-based (F-Ad), and (iv) the children-based filter (F-Children).

Our online algorithm must be able to read a stream of HTTP requests and detect only the small fraction represented by user-URLs. All in a very short period of time, e.g., a few seconds.

We prefer not to process the HTTP request in batches, an approach prone to ``border effect'' problems (requests for the same content split between batches). Therefore, we consider as input a stream of HTTP requests, which are processed using a sliding window. For each HTTP request, we get five fields: $<$*timestamp, URL, Referer, User-Agent, userID*$>$, where the userID is an (anonymized) user identifier, e.g., derived from the client IP address. As an output, our algorithm returns the tuple: $<$*timestamp, URL, referer*$>$.

As described in D4.1, the most accurate filter combination is F-Ref + F-Type + F-Ad. Unfortunately, the F-Ad is based on the humongous Ad-Block catalog, and results too resource-consuming to be implemented online. Thus, we pick the second best candidate, F-Ref + F-Type + F-Children (C $<= 5$), which represents the best trade-off between accuracy and ease of implementation.

The online algorithm is described by the pseudo-code in Alg. 1. It employs a hash table -- the *Candidate Cache*, **C** -- which stores information for every observed Referer, grouped by userID. As soon as a HTTP request is received from the input, we check the presence of the User-Agent in the browser white-list, and verify that the URL and Referer are different. If so, we extract the tuple $<$*timestamp, URL, Referer*$>$. In parallel, we extract the Referer r and we check its presence in **C**. If r is not in **C**, we immediately check the nature of its content with the **F-Type** filter. If r passes the filter, we add it to the candidate cache **C**, that starts updating information about it for a period of time that we call *observation time*, T_O. Such information is: $<$*first timestamp, number of children*$>$, i.e., the timestamp of the first request having r as referer, the number children of r.

Then, we call function `get_user_URL`: for each referer in **C**, we check if its observation time T_O has expired. If so, it means we collected enough information and we can run the heuristics to understand whether such URL may be labeled as user URL or not. If the referer is a user URL, we retrieve its information from the history cache **C**, return it to the output, and delete its entry from **C**.

Algorithm 1 Online user-URL detector.

```
# HTTP Request Stream
Input: HS
# user-URL Stream
Output: IS
# Init Observation Time
1: T_O ← α
# Init Candidate Cache
2: C ← ∅
# Read current HTTP request
3: while h in HS do
4:     h ← timestamp, URL, referer, user-agent, UserID
       # Check User-Agent and URL is different from the referer
5:     if IS_BROWSER(h.user-agent) and h.URL != h.referer then
           # If current referer is not in Candidate Cache
6:         if h.referer ∉ C then
               # If it passes type-based filter
7:             if F_TYPE(h.referer) then
                   # Add referer to the Candidate Cache
8:                 ADD(C, h.referer)
9:                 GET_USER_URL(C)
10:            end if
11:        else
               # Increment the number of children and look for social
               # plugins in the History Cache
12:            UPDATE_INFO(C, h.referer)
13:            GET_USER_URL(C)
14:        end if
15:    end if
16: end while

17: function GET_USER_URL(C)
        # Iterate all referers in the Candidate Cache
18:     for i in C do
            # Check T_O expiration and if it passes children filter
19:         if observation_time(i) > T_O and F_CHILDREN(i) then
                # Retrieve information from C, e.g., the first
                # timestamp
20:             out ← RETRIEVE_INFO(C, i)
                # Send to the output
21:             write(out, IS)
                # Clean structures
22:             remove(i, C)
23:         end if
24:     end for
25: end function
```

2.3.2 Results

Given the substantial modifications on the online version of our user URL detector, we evaluate its accuracy.

As we did for the offline heuristics (see D4.1), also in this case, we evaluate the accuracy of different filters and different parameter tuning on the same dataset, where the ground truth is given by the fact that the dataset has been manually collected. For completeness, we compare again several combinations of filters.

At first, we apply the F-Ref and the F-Type. These two filters together lead to a high recall (98.01%), but also to a low precision (46.22%). By adding the F-Children too, the precision increases, while recall decreases. The results of these experiments are reported in Table 2.1. For our online deployment of this algorithm, we opted for the highest precision which is given by F-Ref + F-Type + F-Children (C $<= 5$).

Method	Recall	Precision	FPR
F-Ref + F-Type	98.01%	46.22%	2.7%
F-Ref + F-Type + F-Children (C <= 1)	98.01%	46.27%	2.7%
F-Ref + F-Type + F-Children (C <= 2)	94.47%	58.56%	1.45 %
F-Ref + F-Type + F-Children (C <= 3)	92.7%	66%	0.95%
F-Ref + F-Type + F-Children (C <= 5)	85.41%	73.4%	0.38%

Table 2.1: Accuracy of the online algorithm.

Method	**Recall**	**Precision**	**FPR**
F-Ref	98.34%	34.0%	4.4%
F-Ref + F-Type	98.34%	46.35%	2.66%
F-Ref + F-Children ($C <= 1$)	94.8%	43.13%	2.84%
F-Ref + F-Children ($C <= 2$)	93.14%	49.76%	2.06%
F-Ref + F-Ad	96.57%	44.14%	2.82%
F-Ref + F-Time ($T < 0.01$s)	97.90%	37.67%	3.7%
F-Ref + F-Time ($T < 0.1$s)	96.13%	41.09%	3.17%
F-Ref + F-Time ($T < 1$s)	87.51%	55.15%	1.39%
F-Ref + F-Time + F-Children ($C <= 1$)	94.80%	58.24%	1.49%
F-Ref + F-Time + F-Children ($C <= 2$)	93.14%	65.39%	1%
F-Ref + F-Time + F-Children ($C <= 3$)	90.38%	68.73%	0.74%
F-Ref + F-Time + F-Children ($C <= 5$)	**83.09%**	**73.58%**	**0.3%**
F-Ref + F-Type + F-Ad + F-Children ($C <= 2$)	91.82%	77.08%	0.45%
F-Ref + F-Type + F-Ad + F-Children ($C <= 3$)	89%	79.1%	0.29%
F-Ref + F-Type + F-Ad	**96.57%**	**66.41%**	**1.17%**

Table 2.2: Accuracy of the offline algorithm.

We ran additional tests to see the impact of increasing the observation time for several values, from 5s up to 30s. We found the best trade-off for $T_O = 15$s, even if the choice has limited impact. We omit results for the sake of brevity. This choice guarantees a stable accuracy, a low memory footprint, and a tight enough processing delay. This is confirmed by very tiny accuracy difference between the online approach (in Table 2.1) and the offline one (in Table 2.2): the online algorithm is basically as accurate as its offline version.

2.4 Quality of Experience for Web browsing

2.4.1 Algorithm design and description

To perform root cause analysis in performance degradation over the users' browsing sessions, at the repository level we exploit statistical measurements and classification techniques over the aggregated data coming from different probes (see Section 2.4.2 of Deliverable 3.1).

The continuous analysis of mean, variance, index of co-variance and other metrics helps the diagnosis algorithm to infer if deviations occur from the usual behavior. Clustering techniques (K-Means) and statistical classification tools like Decision Trees provide further insights on the QoE from different probes to the same web sites, to identify the root cause of the performance degradation.

2.4.2 Results

We are still in the process of forming a statistically significant dataset for evaluating our clustering algorithm.

2.5 Mobile network performance issue cause analysis and multimedia stream measurements

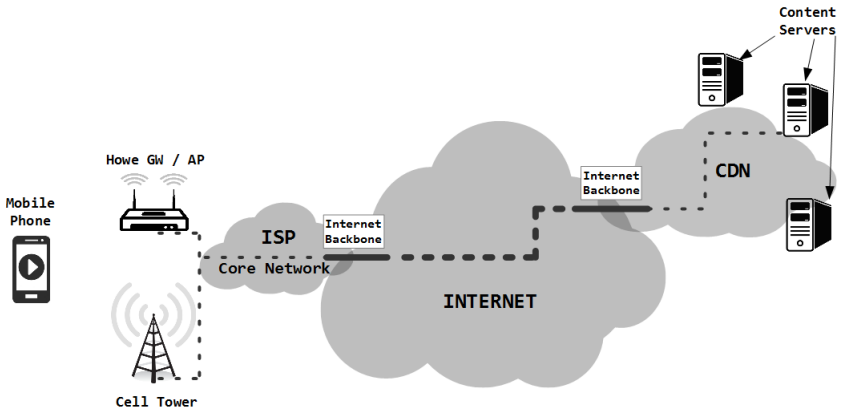


Figure 2.4: Illustration of the segments in a real-world scenario.

In order to detect the types of failures that may cause issues during the video playback, we need to place measurement probes in selected vantage points so that we can extract performance metrics from different segments and devices along the path. To collect measurements from the data receiver and sender's point of view, we place probes at the mobile and the server that correspond to the endpoints of the connection. We add a third probe at the home gateway to act as an intermediate vantage point capable of acquiring metrics from both the local (LAN) and the wide are network (WAN). Here, for simplicity the term WAN encompasses a variety of interconnected networks and devices as can be seen in Figure 2.4.

2.5.1 Algorithm design and description

The correlation between QoE metrics such as stalls during playback and metrics describing the performance of device hardware and network segments is difficult to quantify because of their non-monotonic and some times counter-intuitive relation. Established methods for identifying network or hardware faults do not return information on whether nor how these problems affect the viewer's experience.

For that purpose, we use machine learning methods to learn the correlations between these metrics and QoE metrics and to create a predictive model to detect and characterize the root cause of playback problems.

For the data processing and analysis we use version 3.6.10 of Weka. Weka is a collection of machine learning algorithms and tools for processing, classification, regression and clustering. From the

variety of algorithms offered by Weka we select Decision Trees to perform the classification of the instances. The Decision Trees method builds trees to identify the class each instance belongs to using the concept of mutual information. Our classifier of choice for the data analysis is J48 which is an implementation of the popular C4.5 algorithm offered by Weka.

We collect the dataset from the controlled experiments to establish the ground truth not only for problematic sessions but also for the type of problem that occurred. We afterwards use it for training the problem detection algorithm and evaluate it with the dataset from the real-world measurements. In particular, sessions where re-buffering events have been detected are marked as problematic whereas sessions without buffering as marked as healthy.

The training of the algorithm is performed using the K-fold cross-validation method with *K=10*. This training method breaks the dataset to 10 parts of equal size and uses the 9 parts for training and one for testing. The process is repeated 10 times before producing the final accuracy of the classifier.

We use the filtering tools that are available in Weka to reduce the total number of features by removing irrelevant metrics such as IP addresses, ports and string types like URLs and traffic classification labels. Additionally, we remove all the features that do not variate at all or show very small variance across all the instances. With this approach we are able to reduce the over-fitting problem where the number of parameters is too large relative to the number of instances and results in the reduction of the algorithm's performance.

Overall, we accomplish a 50% reduction of the number of features for the controlled experiments and 38% for the real-world experiments. The lower performance in feature reduction for the real-world dataset is expected given that in real network and usage conditions there is more variance in the system and therefore we cannot remove as many parameters with small or none variance.

2.5.2 Results

All the collected metrics that correspond to a single video session are aggregated to one instance in the dataset. Each instance in the controlled experiments dataset is comprised of 343 metrics out of which there are 113 network metrics for each of the three vantage points, the total number of re-buffering events and from the hardware measurements of the mobile we get the maximum observed CPU utilization, the minimum amount of free memory and the minimum value of the RSSI. For the real-world experiments on the other hand, although all hardware measurements as well as the number of re-buffering events are always available for each instance in the database, the number of network metrics varies depending on the number of vantage points that were used. Therefore, in the final dataset we have instances with either 113 for one vantage point or 226 network parameters when using two.

Before performing the analysis, the instances in the data need to be labelled appropriately in order to be identified and evaluated by the classifier. Specifically, we remove the re-buffering events from the instances in the controlled experiments data and label non problematic instances as `good'. The problematic sessions are labelled as either `lan shaped', `lan congested', `wan shaped', `wan congested', `low rssi', `wifi interference' or `mobile load' according to the simulated scenario they correspond to. In the real-world dataset however we have no knowledge of the type of problem that caused re-buffering at the player side so we are only able to mark the problematic instances as `bad'.

The dataset from the controlled experiments consists of 203 instances in total out of which 129

are labelled as good and 74 as bad. The second dataset obtained from the real-world test, contains 3410 instances that are divided to 2863 good and 547 bad.

Controlled Experiments Analysis
In the results presented in this and the following section apart from the overall accuracy of the algorithm we also use Precision and Recall. Precision is calculated from the ratio of True Positive divided by the sum of True Positives (TP) and False Positives (FP) and it expresses the ratio of the number of relevant instances retrieved to the total number of relevant instances in the dataset. Recall is the ratio of TP divided by the sum of TP and False Negatives (FN). It expresses the ratio of number of relevant instances to the total number of relevant and irrelevant instances retrieved.

$$Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN}$$

Detecting Problems: Firstly, we want to examine weather it is possible to identify problematic video instances through each one of the vantage points or the combination of them. We prepare the data by merging all the labels from problematic instances to a single label `bad', while preserving all good labels. We consecutively evaluate for every vantage point separately and finally with all the points combined. The overall accuracy for the mobile and router is 78.8%, for the server 74.4% and for the combination of all 80.3%. Although the server vantage point is performing worse than the other two when used separately, there is significant improvement when we take measurements from all probes combined.

In Table 2.5.2 we present the performance of the algorithm per vantage point in terms of Precision (P) and Recall (R). We observe that the worse performance of the server derives from the lower accuracy when identifying bad instances. The intuition behind this observation is that most of the problems in our dataset occur far from the server where there is not enough information to correctly identify instances as bad.

	Mobile		Router		Server		Combined	
	P	R	P	R	P	R	P	R
good	0.84	0.82	0.81	0.87	0.79	0.81	0.83	0.87
bad	0.7	0.73	0.73	0.65	0.66	0.62	0.75	0.72
W. Avg.	0.79	0.79	0.78	0.79	0.74	0.74	0.8	0.8

Table 2.3: Accuracy for problem detection in controlled experiments.

Detecting the Location of Problems: In the next step we aim in verifying the algorithm's accuracy when identifying in which part of the data path the problem has occurred. For this purpose we create three new labels `wan', `lan' and `mobile' based on the locality of the problem. In label `wan' we merge wan congestion and wan shaping problems, `lan' contains instances from lan congestion, lan shaping, wifi interference and low rssi scenarios and finally in the `mobile' we place the problematic instances that correspond to mobile load.

The percentage of the correctly classified instances drops to 75.95% in this evaluation case. As expected the accuracy for identifying good instances remains approximately the same as for good/bad classification. However in the related accuracies in Table 2.4, we see that mobile device problems are detected with higher accuracy. This is attributed to the stronger correlation of the hardware metrics with the particular problem.

From the confusion matrix in Table 2.5.2, we see that there are a lot of false positives for the lan meaning that the classifier is incorrectly identifying many problematic instances as good. The poor

classification for the lan label originates from the merge of problems that are quite diverse in nature such as local network and wireless medium faults.

	Precision	Recall
good	0.84	0.87
lan	0.51	0.51
wan	0.6	0.43
mobile	0.9	0.82
Weighted Avg.	0.75	0.76

Table 2.4: Accuracies for localization detection in controlled experiments.

a	b	c	d	classified as
112	10	7	0	a = good
16	25	3	1	b = lan
1	5	12	0	c = wan
1	1	0	9	d = mobile

Table 2.5: Confusion matrix for location detection in controlled experiments.

Detecting the Exact Problem: In the following part of the analysis of the controlled experiments, we train and evaluate the algorithm using all the labels of problematic scenarios that are available in our dataset. In this way we assess the accuracy with which the classifier can detect the root cause behind the problem experienced by the user.

From the output of the classifier we get 73.7% correctly identified instances while different labels are classified with different accuracies as seen in table 2.5.2. In more detail, we observe low performance for lan congestion and 802.11 related problems but much higher for wan shaping and mobile load.

The respective confusion matrix reveals that the classifier performs better when predicting the low rssi and the mobile load labels. This is expected due to the use of hardware metrics that are strong indicators of the device's load and the strength of the signal. We get poor results however when distinguishing between lan congestion and lan shaping problems. This is attributed to the fact that there are many similarities in the network parameters describing the scenarios of shaping and congestion in the local network and therefore the algorithm has trouble distinguishing between the two.

Our next step involves a per-vantage-point evaluation to examine which vantage point is performing better when identifying each problem type. For this evaluation it is necessary to separate the measurements from each point to a different dataset. After the separation, we train and evaluate the classifier for each of the three new datasets.

From the results of each classification we compile Table 2.8 where we compare the precision and recall measures of each vantage point separately and with their combination. From the table we can observe that the vantage point on the mobile is able to detect with higher accuracy problems in the LAN segment and issues of the wireless medium. The router performs well when detecting wan congestion and lan shaping while the only problem the server can identify with better accuracy than the other vantage points is wan shaping. In terms of overall accuracy, the mobile is better than

	Precision	Recall
good	0.84	0.83
lan congested	0.24	0.36
lan shaped	0.67	0.6
wan congested	0.71	0.62
wan shaped	1	0.7
low rssi	0.46	0.6
wifi interference	0.44	0.36
mobile load	0.9	0.82
Weighted Avg.	0.76	0.73

Table 2.6: Accuracies for root-cause detection in controlled experiments.

a	b	c	d	e	f	g	h	classified as
107	11	1	1	0	6	2	1	a = good
7	**5**	2	0	0	0	0	0	b = lan congested
2	2	**6**	0	0	0	0	0	c = lan shaped
2	1	0	**5**	0	0	0	0	d = wan congested
2	0	0	0	**7**	0	1	0	e = wan shaped
2	1	0	0	0	**6**	1	0	f = low rssi
4	1	0	1	0	1	**4**	0	g = wifi interference
1	0	0	0	0	0	1	**9**	h = mobile load

Table 2.7: Confusion matrix for root-cause classification.

the router vantage point which in turn is better than the server, with respective accuracies 73.77%, 69.94% and 68.85%.

The improved accuracy of the mobile, is a strong motivation for instrumenting users' devices. With a single probe collecting measurements from the mobile, the user is able to verify if the problem occurs locally or in a remote part of the network. In the case of a local problem, the algorithm can help the user troubleshoot by providing information about its root cause. If the issue occurs remotely, the user is able to report the problem to the respective network administrator.

Another interesting finding from the results in this section is that any vantage point in our system can tell with good accuracy if a video did not suffer any problems. Based on this insight, middle entities such as service providers can use TCP-driven detectors to detect problems without having to instrument the client or the server.

Finally, we conclude that the usage of a combination of vantage points in a distributed manner, helps to increase the accuracy of the system. Strategically placing more probes on devices along the data path such as edge routers, will not only improve the detection of problems but add knowledge to the system about the location and the nature of the problem.

	MOBILE		ROUTER		SERVER		COMBINED	
	P	R	P	R	P	R	P	R
good	0.83	0.88	0.81	0.85	0.79	0.85	**0.84**	0.83
lan congested	**0.45**	**0.36**	0.31	0.29	0.29	0.29	0.24	0.36
lan shaped	**0.67**	**0.57**	0.4	**0.57**	0.5	0.29	0.67	0.6
wan congested	0.4	0.5	**0.83**	**0.62**	**0.8**	0.5	0.71	0.62
wan shaped	0.5	0.33	0.57	**0.67**	**0.67**	**1**	**0.7**	0.33
mobile load	**1**	**0.73**	0.5	0.27	0.75	0.27	**0.9**	0.82
wifi interference	**0.5**	**0.64**	0.45	0.45	0.43	0.54	0.44	0.36
low rssi	**1**	**1**	**0.4**	0.29	0.25	0.29	0.46	0.6
Weighted Avg.	0.72	0.74	0.69	0.7	0.69	0.76	**0.73**	**0.74**

Table 2.8: Accuracy comparison for all vantage points and labels

2.5.3 Multimedia content delivery in wired provider networks

Considering the scenario depicted in Figure 2.5, there is a clear analogy between the measurements described above and those implemented in monitoring multimedia content delivery performance in a provider's (wired) network.

In this scenario, active probes are programmed to request and receive multimedia streams from a content provider utilizing an ISP's network as transport. These *streaming probes* correspond to the mobile devices in the previous scenario. Additionally, the provider might deploy *line probes* that test actual line bandwidths within the network. These probes correspond to router measurements of the previous scenario. Finally, high-performance *passive probes* identify and measure TCP streams originating from streaming probes towards content provider servers. These measurements correspond to server-based measurements of the previous scenario.

Realizing that measurements of the two scenario map very well to each other, we conclude that the diagnosis algorithm outlined for mobile network performance can be applied easily for multimedia content delivery within ISP networks as well.

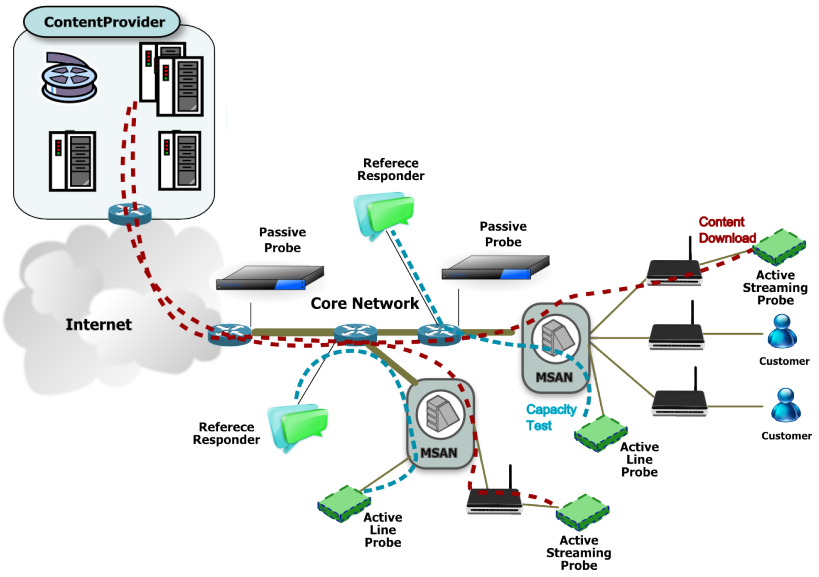


Figure 2.5: Multimedia content delivery measurements in provider networks

2.6 Anomaly detection and root cause analysis in large-scale networks

The goal of the CDN Anomaly Detection (CDN-AD) algorithm is to detect macroscopic anomalies in the aggregate traffic served by CDNs, meaning events that involve multiple flows and/or affect multiple users at the same time. For this purpose, it resorts to the temporal analysis of the entire probability distributions of certain traffic descriptors or features. The proposed statistical non-parametric anomaly detection algorithm works by comparing the current probability distribution of a traffic feature to a set of reference distributions describing its "normal" behavior. At each iteration the algorithm determines a reference for normality by running a reference-set identification sub-routine. The purpose is to find distributions in the recent past (e.g., in an observation window of one or two weeks) which are best suited to represent the current one. In the testing phase the algorithm assesses the statistical compatibility of the current distribution against the distributions included in the reference set. If the current distribution is flagged as anomalous a warning is raised, and it's discharged to be considered as future reference of normality. For further details about the CDN-AD we refer the reader to deliverable D4.1, Section 2.7.2.2, and references therein.

Note that the specific traffic features to be processed by the tool shall capture both the intrinsic and dynamic CDNs mechanisms (e.g., number of flows and bytes served by each CDN server IP address), as well as end-users experienced performance (e.g., flow download throughput). Features distribution are computed on a temporal basis considering time bins of fixed length, referred to as time scale. Time scales is a design parameter that can range from 1 to 60 minutes. Functionally speaking, the algorithm consists of two phases: the training and the detection phase. During the training phase the algorithm accumulates distribution timeseries for a period ranging between 7 and 14 days (depending on the considered timescale). Then, during the detection phase, it uses the information accumulated to identify a suitable reference for normality for the distribution under test. Results of the anomaly detection test, for each traffic dimension, and for each timescale, are logged independently. Further results correlation, for the purpose of the root-cause identification,

is left to the reasoner function and falls beyond the scope of the ADTool implementation.

2.6.1 Algorithm design and description

ADTool is Perl implementation of the aforementioned statistical Anomaly Detection algorithm which runs on top of the *DBStream* streaming data-warehouse system. The software tool requires suitable *DBStream* jobs to compute traffic feature distributions at the required time-scale. It is designed to run online, i.e. it processes the distributions of features as soon as they are available in the *DBStream* views.

ADTool runs iteratively on the output of *DBStream* jobs. At every iteration, the program tries to retrieve the distribution corresponding to the last timebin available and compares it with the distributions in the reference set (i.e. all the distributions corresponding to timebins in a reference window of predefined length).

2.6.1.1 List of Modules

This section provides an overview of the modules the ADTool consists of.

Configs.pm This module provides an interface between the XML configuration file and the rest of the software. The parsing of the XML file is done by the `XML::Simple` Perl standard module.

DataSrc.pm This module provides an interface between the PostgreSQL database used by *DBStream* and the rest of the software. It allows to connect to the database, query for the last available data to compute and write back the output. Both the read and the write interactions with the database are done by the standard Perl DBI module via SQL queries and inserts.

ENKLd.pm This module provides the computation of the normalized Kullback-Leibler divergence between two distribution of values. The two distributions are passed to this module as array references and do not need to be normalized in advance.

RefSet.pm This module defines the package RefSet for managing Reference Sets (collection of past distributions) After being instantiated, a "raw" RefSet object contains all the distributions in the specified reference window. The module provides functions to discard not statistically-relevant distributions (e.g., not enough samples). The output code can be either 2 (if distribution to be tested is too small) or 3 (if the reference set does not contain enough samples).

ADTest.pm This module implements the testing logic of the CDN-AD algorithm. It requires the distribution to be tested, the reference set, and other algorithm parameters (i.e., α, γ). The output code can be either 0 (if normal) or 1 (if anomalous).

2.6.1.2 DBStream Jobs

In order to run *ADTool*, it is firstly necessary to set-up a suitable *DBStream* job to compute counters of the feature for each variable and time bin. The output view of the job should have the following columns:

```
serial_time
<variable name>
<feature name>
```

Note that a single view can be used to collect multiple feature if the `<variable_name>` and the time resolution is compatible.

2.6.1.3 Tool configuration

The configuration of the software is done via an XML file. The available options are:

```
[database] host
[database] port
[database] username
[database] password
[database] features table name (output of DBStream job)
[database] flags table name (output of ADTool)
[analysis] start timestamp
[analysis] end timestamp (0 means run forever)
[analysis] name of variable upon whom the job has computed the distribution
[analysis] feature name
[refset] width (in days)
[refset] guard period (in hours)
[refset] min refset size (minimum number of distributions in refset)
[refset] min distr size (minimum number of samples in distribution)
[refset] m (number of top ranked distributions in refset)
[refset] k (currently unused)
[ADtest] alpha (algorithm's sensitivity)
```

A sample configuration file is:

```
<ADTool_config>
<!-- ********************************************************************************************
      task description
******************************************************************************************** -->
<Description>adtool on youtube (ip,imsi_cnt)</Description>

<!-- ********************************************************************************************
                            settings for database connection
******************************************************************************************** -->
<Database host="localhost" port="5440" dbname="dbstream" user="dbstream" password="FT4hhyhL" >
<features_table>adtool_mw14_gg11_youtube_features_serverip_600</features_table>
```

```
<flags_table>adtool_youtube_flags_serverip_600</flags_table>
</Database>

<!-- ***************************************************************************************
               analysis settings (time span, granularity, feature name, etc.)
**************************************************************************************** -->
<Analysis>
<start>1396648800</start><!-- beginning of analysis -->
<end>0</end><!-- end of analysis, 0 means run online -->
<granularity>600</granularity><!-- time granularity in seconds -->
<variable>server_ip</variable><!-- variable of the distributions -->
<feature>imsi_cnt</feature><!-- name of the traffic feature -->
</Analysis>

<!-- ***************************************************************************************
                                settings for reference set
**************************************************************************************** -->
<RefSet>
<width>7</width><!-- reference set time window in days -->
<guard>2</guard><!-- guard period in hours -->

<min_distr_size>100</min_distr_size><!-- min number of samples in distributions -->
<min_refset_size>80</min_refset_size><!-- min number of distributions in refset -->
<slack_var>0.1</slack_var><!-- for comparing size of timebins -->
<m>50</m><!-- usually ~1/4 min_refset_size -->
<k>2</k><!-- number of clusters for pruning --> <!-- currently unused -->
</RefSet>

<!-- ***************************************************************************************
                                     settings for AD test
**************************************************************************************** -->
<ADTest>
<alpha>0.05</alpha><!-- sensitivity -->
</ADTest>
</ADTool_config
```

2.6.1.4 Workflow

The logic is defined in the main executable `adtool.pl`. The arguments for running the program are:
`--config␣<XML_CONFIG_FILE>`
`--log␣<LOG_FILE>`

The execution workflow of /AdTool| is described in Figure 2.6.

2.6.1.5 Output

Upon completion of each iteration, the output is reported on `STDOUT` as well as on the database's flag table specified in the configuration. For each iteration running on a time-bin, the row inserted in the flag table is composed by the following column:

- beginning timestamp of the timebin
- feature name

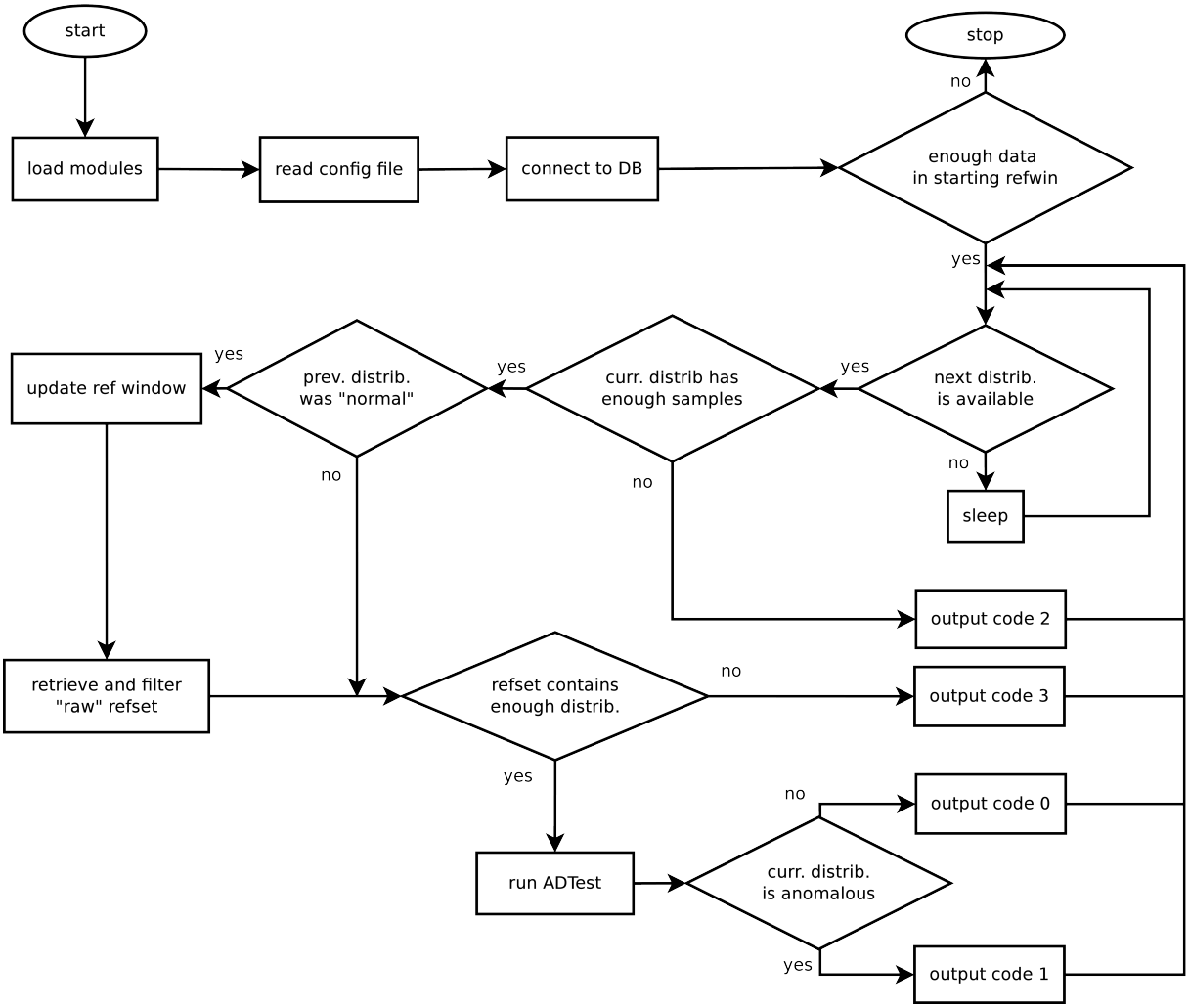


Figure 2.6: Flowchart of the *ADTool*.

- output code $(0, 1, 2, 3, 4)$
- score
- γ
- Φ_α
- Output codes:
- 0: distribution is ``normal''
- 1: distribution is anomalous
- 2: distribution does not contain enough samples
- 3: refset does not contain enough distributions.
- 4: currently unused

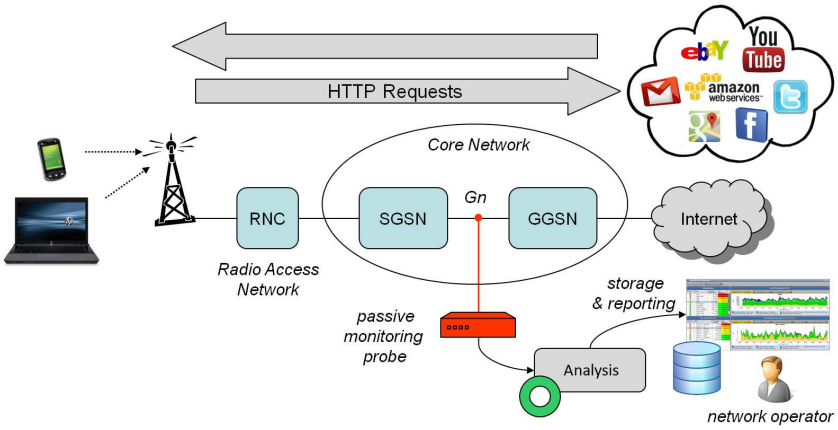


Figure 2.7: Passive traffic analysis in an operational 3G Network. The ADTool analyzes the flows observed at the Gn interface.

KPI	code 0 (A-free)	code 1 (Anomaly)	code 2	code 3
Vol. per IP	81.5%	13.4%	1.5%	3.6%
Users per IP	84.9%	9.1%	1.5%	4.5%

Table 2.9: ADTool results for distribution of volume and number of users per server IP in a 3G network, for a full month.

2.6.2 Results

The *ADTool* is currently running on-line on top of an operational 3G network. Packets are captured on the Gn interface links between the GGSN and SGSN nodes, using a passive monitoring probe, as depicted in figure 2.7.

Here we report some preliminary results of the outputs generated by the *ADTool* during a complete month. In this evaluation, we consider two different KPIs for anomaly detection: the distribution of downloaded traffic volume per server IP address, and the distribution of number of unique users per server IP address. The rationale behind these two features is that CDN cache selection policies, flash-crowd events, or even misbehaving user terminals may potentially cause a shift in the aforementioned distributions worth to be detected.

During the first week of the evaluation, the *ADTool* uses the monitored data for training purposes, building the reference set of normal behavior distributions, for both KPIs. Table 2.9 reports the obtained results for the full evaluation month, reporting the fraction of generated outputs, according to the aforementioned output codes. The results of each analysis are reported every 10 minutes, and the total number of outputs is therefore about 4500.

While the majority of the samples correspond to anomaly free outputs, there is about one out of ten outputs reporting an anomaly. These results are preliminary and we do not include in this discussion an evaluation of the detection accuracy or the false alarms generation. Indeed, the main contribution of these results is to acknowledge that the algorithm is currently running and is able to process all the traffic captured on a national wide mobile network, running on top of DBStream. In the next section, and to complete the overall image on how DBStream is able to cope with large amounts of monitoring data, we report an evaluation of its performance as compared to a traditional big data parallel solution based on the MapReduce paradigm.

2.7 Anomaly detection and root cause analysis in large-scale networks: data mining algorithms

Here we describe the application and performance evaluation of a cloud-based approach, named SEARUM, to efficiently mine association rules on a distributed computing model. SEARUM has been applied to mPlane network traffic traces for exploratory data analysis and consists of a series of distributed MapReduce jobs run in the cloud.

The automatic analysis of huge network traffic data is a challenging and promising task. Association rule mining is an exploratory data analysis method able to discover interesting and hidden correlations among data. It is a two-step process: (i) Frequent itemset extraction and (ii) association rule generation from frequent itemsets. Since the first phase represents the most computationally intensive knowledge extraction task, effective solutions have been widely investigated to parallelize the itemset mining process both on multi-core processors and with a distributed architecture. However, when a large set of frequent itemsets is extracted, the generation of association rules from this set becomes a critical task.

The challenge is twofold: (i) this data mining process is characterized by computationally intensive tasks, thus requiring efficient distributed approaches to increase its scalability, and (ii) its results must add value to the domain expert knowledge.

In the context of the mPlane project we designed, developed and applied a horizontally-scalable approach.

2.7.1 Algorithm design and description

As introduced in deliverable D3.1, SEARUM consists of a series of distributed jobs run in the cloud. Each job receives as input the result of one or more preceding jobs and performs one of the steps required for association rule mining. Currently, each job is performed by one or more MapReduce tasks run on a Hadoop cluster.

The SEARUM architecture contains the following jobs:

- Network measurement acquisition
- Data pre-processing
- Item frequency computation
- Itemset mining
- Rule extraction
- Rule aggregation and sorting

A complete description of the architecture is provided in deliverable D4.2. In the follows we evaluate SEARUM in a cloud-based distributed environment.

2.7.2 Results

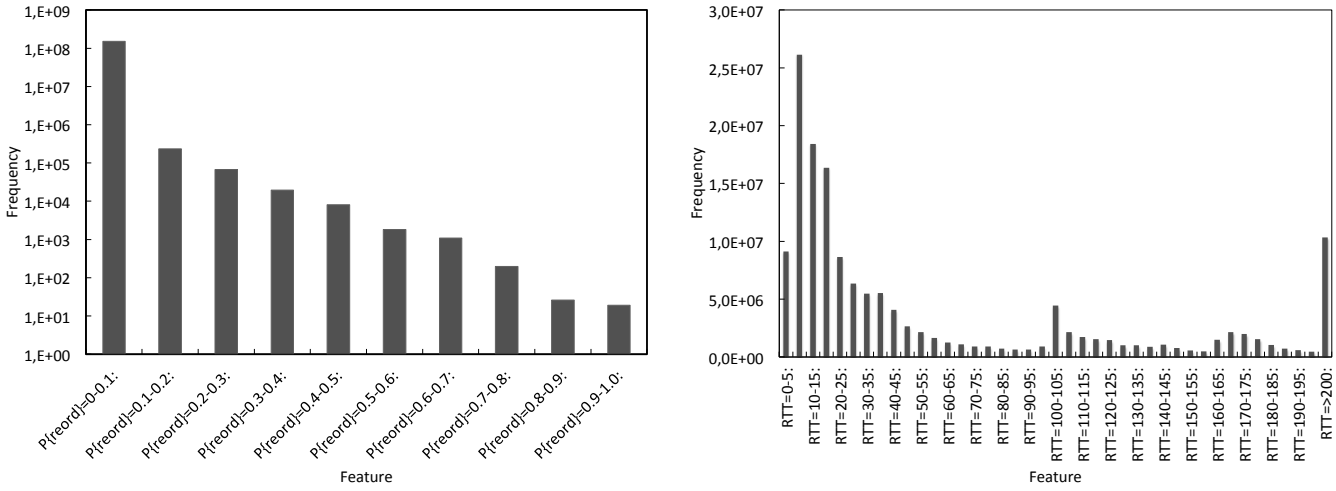
A set of preliminary experiments have been performed analyzing SEARUM behavior on real mPlane datasets. We assessed (i) the performance of the association rule mining, (ii) the network knowledge characterization and (iii) the number of extracted association rules by varying the support and confidence thresholds (Section 2.7.2),

SEARUM has been applied to two real datasets. We will refer to each dataset as D1 or D2 as shown in Table 2.10, where the number of TCP flows and the size of each dataset are also reported.

Network knowledge characterization

Dataset	Number of TCP flows	Size [Gbyte]
D1	11,325,006	5.28
D2	413,012,989	192.56

Table 2.10: Network traffic datasets



(a) Item distribution for the $P\{reord\}$ feature

(b) Item distribution for the RTT feature

Figure 2.8: Dataset D2

We evaluated the effectiveness of the proposed approach on real network traffic traces. In particular, we analyzed: (i) the usefulness of the extracted association rules in supporting the knowledge discovery process, and (ii) the item frequency distribution.

As example, the following two rules R1 and R2 are generated from dataset D1 and D2, respectively. Both rule have high confidence values and lift greater than 1 (rule support, confidence, and lift are reported in brackets after each rule).

$R_1 : \{Port = 80, P\{reord\} = 0 - 0.1, DataPkt = 1 - 2, DataBytes = 4 - 5\} \rightarrow Class = HTTP$ (0.313, 0.999, 1.765)

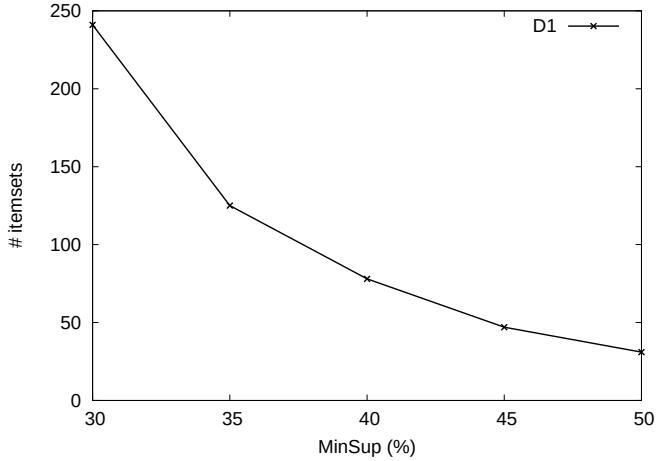
$R_2 : \{P\{dup\} = 0 - 0.1, NumPkt \leq 1, DataPkt \leq 1, Class = SSL\} \rightarrow Port = 443$ (0.013, 0.993, 4.944)

Based on rule R_1, the HTTP protocol is mainly used to transmit a set of TCP flows sent by the server through the TPC port 80. For these flows, the number of packets is in the range 10÷100 and a large number of bytes is transmitted (from 10,000 to 100,000). These flows can be generated when very large files are downloaded (e.g., YouTube videos).

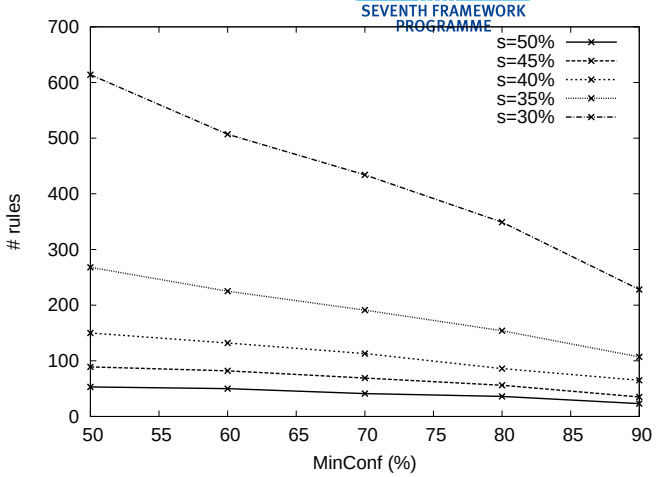
Rule R_2 reports that the TCP $Port$ 443 (HTTPS) is mainly used to transmit flows with SSL/TLS coded protocol and less than 10 packets. These flows can be generated when logging into websites through a secure connection (e.g., Facebook, Twitter).

We also analyzed the item frequency distribution to characterize the network activity. Figure 2.8 considers the Round-Trip-Time (RTT) and the flow reordering probability ($P\{reord\}$), which are discussed as representative features.

The item distribution for the $P\{reord\}$ feature is characterized by a very frequent item which models most TCP flows: they have a very low $P\{reord\}$, i.e., from 0 to 0.1. This data distribution analyzed over time and for different (sub)networks may be exploited to identify periods of time or (sub)networks that become less reliable or whose packets change path more frequently than usual.



(a) Number of extracted itemsets for different $MinSup$ values



(b) Number of extracted rules for different values of $MinConf$ and $MinSup$

Figure 2.9: Dataset D1: Effect of $MinSup$ and $MinConf$ thresholds

The item distribution for the RTT, instead, shows four peaks:

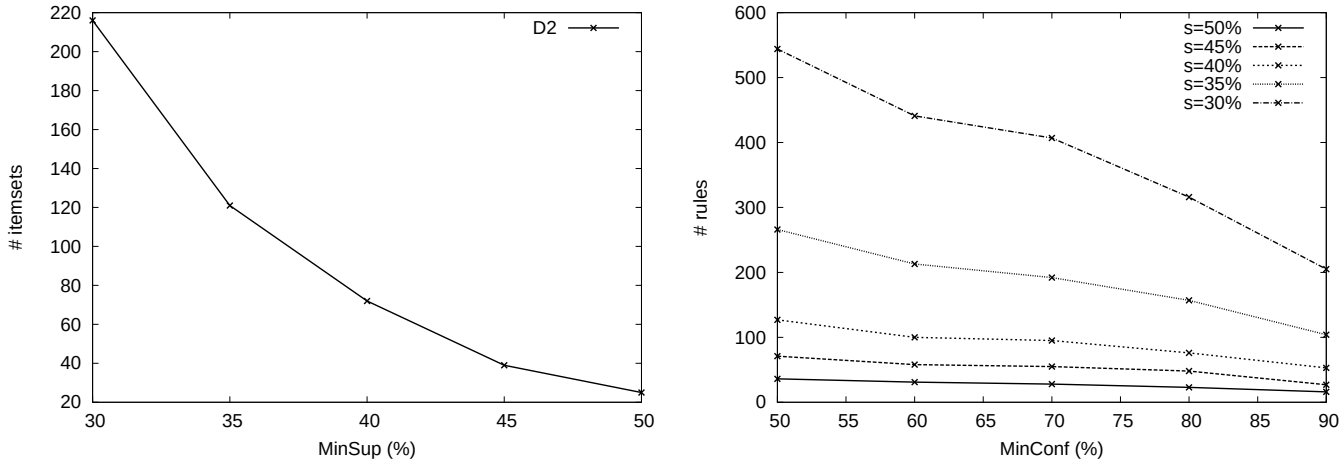
- the first peak around 5-20 ms may represent local network traffic
- the second peak around 100 ms may represent external traffic inside the same ISP or in the same geographical zone (e.g., country, continent)
- the third peak around 170 ms may represent traffic towards long-distance destinations (e.g., other continents)
- finally, the last peak over 200 ms may represent network problems or unresponsive services

Effect of the support and confidence thresholds

Minimum support ($MinSup$) and confidence ($MinConf$) thresholds significantly affect the number of extracted itemsets and association rules.

When decreasing the $MinSup$ value, the number of frequent itemsets grows non linearly and the complexity of the frequent itemset extraction task significantly increases. High $MinConf$ values represent a tighter constraint on rule selection. Consequently, when increasing $MinConf$ less rules are mined, but these rules tend to represent stronger correlations among data. High $MinConf$ values should be often combined with low $MinSup$ values to lead the extraction of peculiar (i.e., not very frequent) but highly correlated rules.

Figures 2.9(a) and 2.10(a) plot, for the two reference datasets, the number of extracted itemsets when varying $MinSup$. Figures 2.9(b) and 2.10(b) report the number of association rules for different $MinConf$ values.



(a) Number of extracted itemsets for different values of $MinSup$ (b) Number of extracted rules for different values of $MinConf$ and $MinSup$

Figure 2.10: Dataset D2: Effect of $MinSup$ and $MinConf$ thresholds

3 Distributed computing platforms: tools and performance

In this chapter we report on the tools being developed at the repository level to schedule jobs on distributed computing platforms, with their design and results.

Tools described in this chapter have been made available by mPlane partners and they can be accessed at `http://www.ict-mplane.eu/public/software`.

3.1 `Hadoop Fair Sojourn Protocol`: a scheduler for Apache Hadoop

This section presents the design of Hadoop Fair Sojourn Protocol, a size-based scheduler for Apache Hadoop, developed jointly with partners of the EU-project BigFoot.

A key problem for the applicability of size-based scheduling in general is that job size is most often not known exactly *a priori*: it can rather be estimated.

In Section 3.1.1, we describe a simulation-based evaluation of size-based schedulers in presence of errors; our result are promising, since we show that size-based scheduling performs well when the job sizes are not extremely skewed; when the skew is large, on the other hand, we show that simple adaptations are sufficient to obtain close to optimal scheduling in most cases.

These results encourage us to implement size-based scheduling in Hadoop. In Section 3.1.2, we describe Hadoop Fair Sojourn Protocol (HFSP), our approach to this problem. In context of heterogeneous workloads, as the ones expected in mPlane, HFSP performs excellently.

A scheduler such as HFSP relies on the concept of *job preemption* to release resources for jobs with higher priorities: waiting for running tasks to complete is not optimal, since it will result in higher latency for high-priority tasks. In Section 3.1.3, we describe our implementation of a *suspend* primitive that uses operating system signaling to stop low-priority tasks, and resumes them when high-priority ones are completed.

3.1.1 Revisiting Scheduling Based On Estimated Job Sizes

We now begin our exploration of the investigated scheduling issues with a simulation-based analysis of the behavior of size-based scheduling protocols in the presence of errors: these results guide and motivate us in the implementation of the Hadoop HFSP scheduler, described in Section 3.1.2. Here, we synthesize a submitted work which is available as a pre-print [19]: we refer the interested reader to that work for more detail.

3.1.1.1 Background

We now introduce the SRPT and FSP size-based scheduling protocols, and describe the effects that estimation errors have on their behavior, focusing on the difference between over- and under-estimation. We notice that under-estimation triggers a behavior which is problematic in particular for heavy-tailed job size distributions, and we propose a solution to handle it.

First In First Out (FIFO) and Processor Sharing (PS) are arguably the two most simple and ubiquitous scheduling disciplines in use in many systems; for instance, the FIFO and FAIR schedulers in Hadoop are inspired by these two approaches. In FIFO, jobs are scheduled in the order of their submission, while in PS resources are divided evenly so that each active jobs keeps progressing. In loaded systems, these disciplines have severe shortcomings: in FIFO, large running jobs can delay very significantly small ones that are waiting to be executed; in PS, each additional job delays the completion of *all* the others.

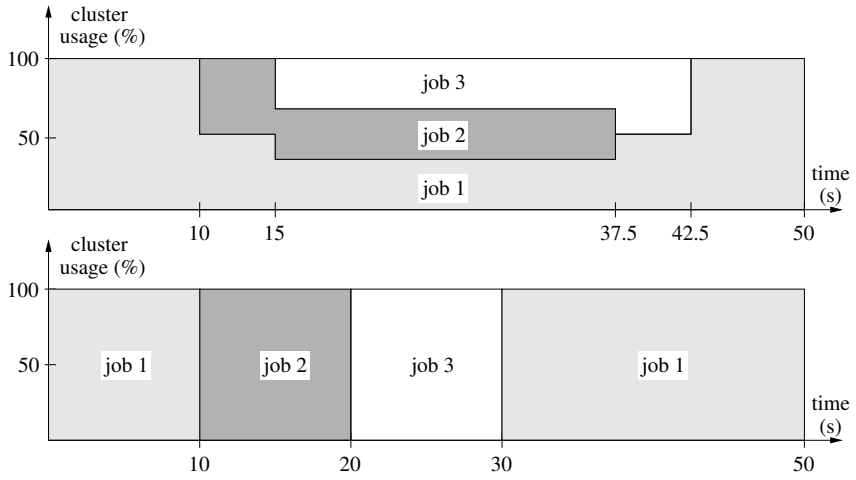


Figure 3.1: Comparison between PS (top) and SRPT (bottom).

SRPT Essentially, size-based scheduling adopts the idea of giving priority to small jobs: as such, they will not be slowed down by large ones. The Shortest Remaining Processing Time (SRPT) policy, which prioritizes jobs that need the least amount of work to complete, is the one that minimizes the mean *sojourn time* (or *response time*), that is the time that passes between a job submission and its completion [40].

Figure 3.1 compares PS with the SRPT scheduling discipline with an illustrative example: in this case, two small jobs -- j_2 and j_3 -- are submitted while a large job j_1 is running. While in PS the three jobs run (slowly) in parallel, in a size-based discipline j_1 is *preempted*: the result is that j_2 and j_3 complete earlier. It is worth noting that, in this case, the completion time of j_1 does not suffer from preemption: somewhat counter to intuition, this is often the case for SRPT-based scheduling [28].

FSP SRPT may cause *starvation* (i.e., never providing access to resources): for example, if small jobs are constantly submitted, large jobs may never get served. FSP (also known in literature as *fair queuing* [34] and *Vifi* [26]) is a policy that doesn't suffer from starvation by virtue of *job aging*, i.e., gradually increasing the priority of jobs that are not scheduled. More precisely, FSP serves the job that would complete earlier in a *virtual* emulated system running a processor sharing (PS) discipline: since all jobs eventually complete in the virtual system, they will also eventually be scheduled in the real one.

In the absence of errors, a policy such as FSP is particularly desirable because it obtains a value of MST which is close to what is provided by SRPT while guaranteeing a strong notion of fairness in the sense that FSP *dominates* PS: no jobs complete later in FSP than in PS [22]. When errors are present, such a property cannot be guaranteed; however, as our experimental results in Section 3.1.1.5 show, FSP still preserves better fairness than SRPT even when errors are present.

3.1.1.2 Dealing With Errors: SRPTE and FSPE

We now consider the behavior of SRPT and FSP when the scheduler has access to *estimated* job sizes rather than exact ones. For clarity, we will refer hereinafter to *SRPTE* and *FSPE* in this case.

In Figure 3.2, we provide an illustrative example where a single job size is over- or under-estimated while the others are estimated correctly, focusing (because of its simplicity) on the behavior of SRPTE; job sojourn times are represented by the horizontal arrows. The left column of Figure 3.2 illustrates the effect of over-estimation. In the top, we show how the scheduler behaves without errors, while in the bottom we show what happens when the size of job $J1$ is over-estimated. The graphs shows the remaining (estimated) processing time of the jobs over time (assuming a normalized service rate of 1). Without errors, jobs $J2$ does not preempt $J1$, and $J3$ does not preempt $J2$. Instead, when the size of $J1$ is over-estimated, both $J2$ and

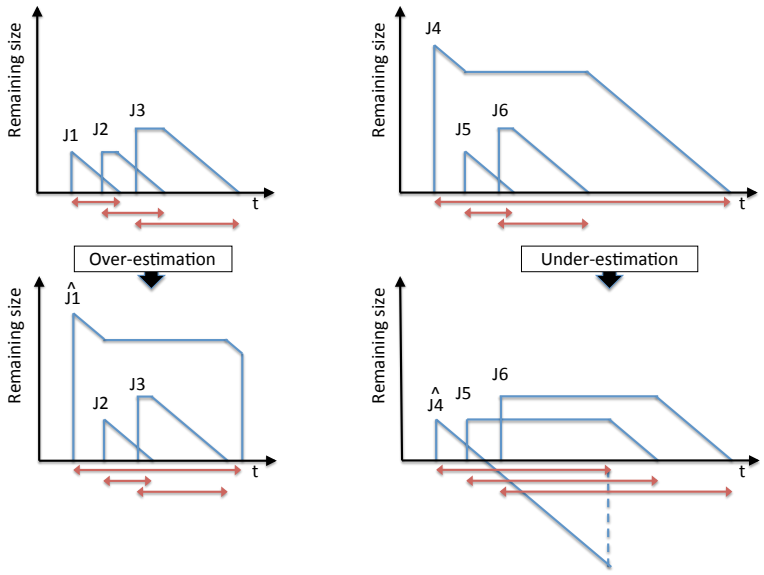


Figure 3.2: Examples for job under- and over-estimation.

$J3$ preempt $J1$. Therefore, the only job suffering (i.e., experiencing higher sojourn time) is the one that has been over-estimated. Jobs with smaller sizes are always able to preempt an over-estimated job, therefore the basic property of SRPT (favoring small jobs) is not significantly compromised.

The right column of Figure 3.2 illustrates the effect of under-estimation. With no estimation errors (top), a large job, $J4$, is preempted by small ones ($J5$ and $J6$). If the size of the large job is under-estimated (bottom), its estimated remaining processing time eventually reaches zero: we call *late* a job with zero or negative estimated remaining processing time. *A late job cannot be preempted* by newly arrived jobs, since their size estimation will always be larger than zero. In practice, since preemption is inhibited, the under-estimated job *blocks the system* until the end of its service, with a negative impact on multiple waiting jobs.

This phenomenon is particularly harmful when job sizes are heavily skewed: if the workload has few very large jobs and many small ones, a single late large job can significantly delay several small ones, which will need to wait for the late job to complete before having an opportunity of being served.

Even if the impact of under-estimation seems straightforward to understand, surprisingly no work in the literature has ever discussed it. To the best of our knowledge, we are the first to identify this problem, which significantly influences scheduling policies dealing with inaccurate job size.

In FSPE, the phenomena we observe are analogous: job size over-estimation delays only the over-estimated job; under-estimation can result in jobs terminating in the virtual PS queue before than in the real system; this is impossible in absence of errors due to the dominance of FSP over PS. We therefore define *late* jobs in FSPE as those whose execution is completed in the virtual system but not yet in the real one and we notice that, analogously to SRPTE, also in FSPE late jobs can never be preempted by new ones, and they block the system until they are all completed.

3.1.1.3 Our Solution

Now that we have identified the issue with existing size-based scheduling policies, we propose our counter-measure. Several alternatives are envisionable, including for example updating job size estimations if new information becomes available as work progresses: such a solution may not however be always feasible, due to limitations in terms of information or computational resources available to the scheduler.

We propose, instead, a simple solution that requires no additional job size estimation, based on the simple

idea that *late jobs should not prevent executing other ones*. This goal is achievable with a variety of techniques having in common the property that the scheduler takes corrective actions when one or more jobs are late, guaranteeing that -- even when very large late jobs are being executed -- newly arrived small jobs will get executed soon.

We show here *FSPE+PS*, which is a simple modification to FSPE: the only difference is that, when one or more jobs are late, (i.e., they have completed in the emulated virtual system and not in the real one), *all late jobs are scheduled concurrently in a PS fashion*. FSPE+PS inherits from FSP and FSPE the guarantee that starvation is absent, it is essentially as complex to implement as FSP is and, as we show in Section 3.1.1.5, it performs close to optimally in most experimental settings we observe. We remark that due to the dominance of FSP with respect to PS, if there are no size estimation errors no jobs can ever become late: therefore, with no error FSPE+PS is equivalent to FSP.

Several alternatives to FSPE+PS are possible: we experimented for example with similar policies that are based on SRPT rather than on FSP, that use a least-attained-service policy rather than a PS one for late jobs, and/or that schedule aggressively jobs that are not late yet as soon as at least one reaches the ``late'' stage. With respect to the metrics we use in this work, their behavior is very similar to the one of FSPE+PS, and for reasons of conciseness we do not report about them here. We however encourage the interested reader to examine their implementation.[1]

3.1.1.4 Evaluation Methodology

Understanding size-based scheduling systems when there are estimation errors is not a simple task. The complexity of the system makes an analytical study feasible only if strong assumptions, such as a bounded error [49], are imposed. Moreover, to the best of our knowledge, no analytic model for FSP (without estimation error) is available, making an analytic evaluation of FSPE and FSPE+PS even more difficult.

For these reasons, we evaluate our proposed scheduling policies through simulation. The simulative approach is extremely flexible, allowing to take into account several parameters -- distribution of the arrival times, of the job sizes, of the errors. Previous simulative studies (e.g., [33]) have focused on a subset of these parameters, and in some cases they have used real traces. We developed a tool that is able to both reproduce real traces and generate synthetic ones. Moreover, thanks to the efficiency of the implementation, we were able to run an extensive evaluation campaign, exploring a large parameter space. For these reasons, we are able to provide a broad view of the applicability of size-based scheduling policies, and show the benefits and the robustness of our solution with respect to the existing ones.

Scheduling Policies Under Evaluation In this work, we take into account different scheduling policies, both size-based and blind to size. For the size-based disciplines, we consider SRPT as a reference for its optimality with respect to the MST. When introducing the errors, we evaluate SRPTE, FSPE and our proposal, FSPE+PS, described in Section 3.1.1.1.

For the scheduling policies blind to size, we have implemented the *First In, First Out* (FIFO) and *Processor Sharing* (PS) disciplines. These policies are the default disciplines used in many scheduling systems -- e.g., the default scheduler in Hadoop [47] implements a FIFO policy, while Hadoop's FAIR scheduler is inspired by PS; the Apache web server delegates scheduling to the Linux kernel, which in turn implements a PS-like strategy [41]. Since PS scheduling divides evenly the resources among running jobs, it is generally considered as a reference for its fairness. Finally, we consider also the *Least Attained Service* (LAS) [39] policy. LAS scheduling, also known in the literature as *Foreground-Background* (FB) [31] and *Shortest Elapsed Time* (SET) [17], is a preemptive policy that gives service to the job that has received the least service, sharing it equally in a PS mode in case of ties. LAS scheduling has been designed considering the case of heavy-tailed job size distributions, where a large percentage of the total work performed in the system is due to few very large jobs, since it gives more priority to small jobs than what PS would do.

[1] `http://bit.ly/schedulers`

Table 3.1: Simulation parameters.

Parameter	Explanation	Default
sigma	σ in the log-normal error distribution	0.5
shape	shape for Weibull job size distribution	0.25
timeshape	shape for Weibull inter-arrival time	1
njobs	number of jobs in a workload	10,000
load	system load	0.9

Performance Metrics In this document, we evaluate scheduling policies according to *mean sojourn time* (MST). MST is the time that passes between the moment a job is submitted and when it completes; such a metric is widely used in the scheduling literature.

Fairness is instead a more elusive concept: in his survey on the topic, Wierman affirms that ``*fairness is an amorphous concept that is nearly impossible to define in a universal way*'' [48]. We refer to our full work [19] for an evaluation of fairness based on *slowdown*, i.e., the ratio between a job's sojourn time and its size.

Parameter Settings Our goal is to empirically evaluate scheduling policies in a wide spectrum of cases. Table 3.1 synthetize the parameters that our simulator can accept as inputs; they are explained in detail in the following. In this synthesis, we provide our results obtained by varying shape and sigma only; other parameters are less fundamental and results are covered in our full work [19].

Job Size Distribution: Job sizes are generated according to a Weibull distribution, which allows us to evaluate both heavy-tailed and light-tailed job size distributions. Indeed, the ***shape*** parameter allows to interpolate between heavy-tailed distributions (shape < 1), the exponential distribution (shape$= 1$), the Raleigh distribution (shape $= 2$) and bell-shaped distributions centered around the `1' value (shape > 2). We set the *scale* parameter of the distribution to ensure that its mean is 1.

Since scheduling problems have been generally analyzed on heavy-tailed workloads with job sizes using distributions such as Pareto, we consider a default heavy-tailed case of shape $= 0.25$. In our experiments, we vary the shape parameter between a very skewed distribution with shape $= 0.125$ and a bell-shaped distribution with shape $= 4$.

Size Error Distribution: We consider log-normally distributed error values. A job having size s will be estimated as $\hat{s} = sX$, where X is a random variable with distribution

$$\text{Log-}\mathcal{N}(0, \sigma^2). \tag{3.1}$$

This choice satisfies two properties: first, since error is multiplicative, the absolute error $\hat{s} - s$ is proportional to the job size s; second, under-estimation and over-estimation are equally likely, and for any σ and any factor $k > 1$ the (non-zero) probability of under-estimating $\hat{s} \leq \frac{s}{k}$ is the same of over-estimating $\hat{s} \geq ks$. This choice also is substantiated by empirical results: in our implementation of the HFSP scheduler for Hadoop [36], we found that the empirical error distribution was indeed fitting a log-normal distribution.

The ***sigma*** parameter controls σ in Equation 3.1, with a default -- used if no other information is given -- of 0.5; with this value, the median factor k reflecting relative error is 1.40. In our experiments, we let sigma vary between 0.125 (median k is 1.088) and 4 (median k is 14.85).

It is possible to compute the correlation between the estimated and real size as σ varies. In particular, when sigma is equal to 0.5, 1.0, 2.0 and 4.0, the correlation coefficient is equal to 0.9, 0.6, 0.15 and 0.05 respectively.

The mean of this distribution is always larger than 1, and growing as sigma grows: the system is biased towards overestimating the aggregate size of several jobs, limiting the underestimation problems that FSPE+PS is designed to solve. Even in this setting, the results in Section 3.1.1.5 show that the improvements obtained by using FSPE+PS are still apparent.

Job Arrival Time Distribution: For the job inter-arrival time distribution, we use a Weibull distribution for its flexibility to model heavy-tailed, memoryless and light-tailed distributions. We set the default of its shape parameter (***timeshape***) to 1, corresponding to ``standard'' exponentially distributed arrivals.

Other Parameters: The ***load*** parameter is the mean arrival rate divided by the mean service rate. As default value, we use the same value of 0.9 used by Lu et al. [33].

The number of jobs (***njobs***) in each simulation round is 10,000 (in additional experiments -- not shown for space reasons -- we varied this parameter, without obtaining significant differences). For each experiment, we perform at least 30 repetitions, and we compute the confidence interval for a confidence level of 95%. For very heavy-tailed job size distributions (shape ≤ 0.25), results are very variable and therefore, in order to obtain stable averages, we performed hundreds and/or thousands of experiment runs, until the confidence levels have reached the 5% of the estimated value.

Simulator Implementation Details Our simulator is available under the Apache V2 license.[2] It has been conceived with ease of prototyping in mind: for example, our implementation of FSPE as described in Section 3.1.1.1 requires 53 lines of code. Workloads can be both replayed from real traces and generated synthetically.

The simulator has been written with a focus on computational efficiency. It is implemented using an event-based paradigm, and we used efficient data structures based on B-trees.[3] As a result of these choices, a ``default'' workload of 10,000 jobs is simulated in around half a second, while using a single core in our machine with an Intel T7700 CPU. We use IEEE 754 double-precision floating point values for the internal representation of time and job sizes.

3.1.1.5 Experimental Results

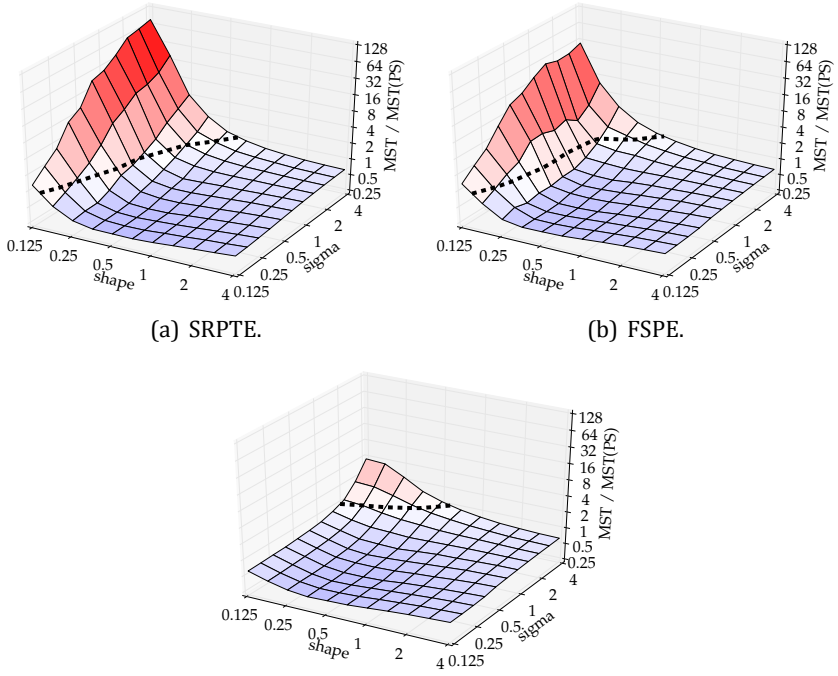
We now present our experimental findings. For all the results shown in the following, the parameters whose values are not explicitly stated take the default values shown in Table 3.1. For the readability of the figures, we do not show the confidence intervals: for all the points, in fact, we have performed a number of runs sufficiently high to obtain a confidence interval smaller than 5% of the estimated value. We first present our results on synthetic workloads generated according to the methodology of Section 3.1.1.4; we then show the results by replaying two real-world traces from workloads of Hadoop and of a Web cache.

Additional results, proving that these results apply also to real workloads, are available in our full work [19].

Mean Sojourn Time Against PS We begin our analysis by comparing the performance of the three size-based scheduling policies, using PS as a baseline because PS and its variants are the most widely used set of scheduling policies in real systems. In Figure 3.3 we plot the value of the MST obtained using respectively SRPTE, FSPE and FSPE+PS, normalizing it against the MST of PS. We vary the sigma and shape parameters influencing respectively job size distribution and error rate; we will see that these two parameters are the ones that influence performance the most. Values lower than one (below the dashed line in the plot) represent regions where size-based schedulers outperform PS.

In accordance with intuition and to what is known from the literature, we observe that the performance of size-based scheduling policies depends on the accuracy of job size estimation: as sigma grows, performance suffers. From Figures 3.3(a) and 3.3(b), we however observe a new phenomenon: *job size distribution impacts performance even more than size estimation error.* On the one hand, we notice that large areas of the plots (shape > 0.5) are almost insensitive to estimation errors; on the other hand, we see that MST becomes very large as job size skew grows (shape < 0.25). We attribute this latter phenomenon to the fact that, as we highlight in Section 3.1.1.1, late jobs whose estimated remaining (virtual) size reaches zero are never preempted. If a large job is underestimated and becomes ``late'' with respect to its estimation, small jobs will

[2] `https://bitbucket.org/bigfootproject/schedsim`
[3] `http://stutzbachenterprises.com/blist/`



(a) SRPTE.

(b) FSPE.

(c) FSPE+PS.

Figure 3.3: Mean sojourn time against PS.

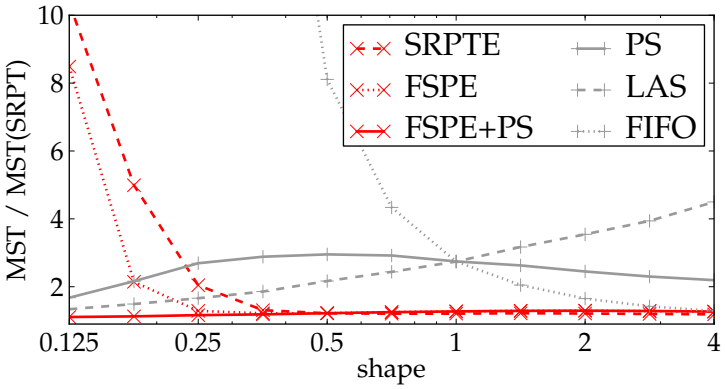


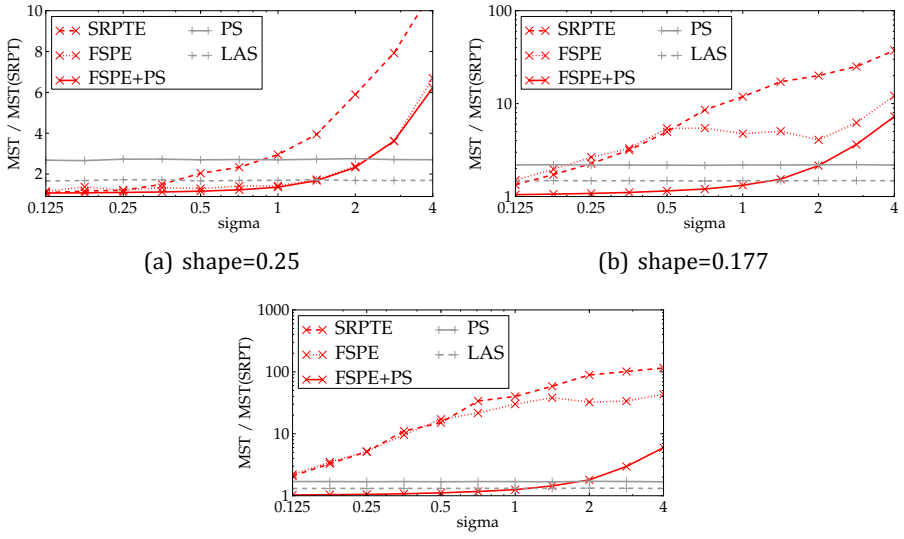
Figure 3.4: Impact of shape.

have to wait for it to finish in order to be served; when job distribution is heavy tailed, this results in large delays whenever one of the biggest jobs is underestimated.

As we see with Figure 3.3(c), *FSPE+PS outperforms PS in a large class of heavy-tailed workloads* where SRPTE and FSPE suffer. The net result is that a size-based policy such as FSPE+PS is outperformed by PS only in extreme cases where the job size distribution is extremely skewed *and* job size estimation is very imprecise.

It may appear surprising that, when job size skew is not extreme, size-based scheduling can outperform PS even when size estimation is very imprecise: even a small correlation between job size and its estimation can direct the scheduler towards choices that are beneficial on aggregate. In fact, as we see more in detail in the following, sub-optimal scheduling choices become less penalized as the job size skew diminishes.

Impact of shape We now delve into details and examine how schedulers perform when compared to the optimal MST that SRPT obtains. In the following Figures, we show the ratio between the MST obtained with



(a) shape=0.25

(b) shape=0.177

(c) shape=0.125

Figure 3.5: Impact of error on heavy-tailed workloads, sorted by growing skew.

the scheduling policies we implemented and the optimal one of SRPT.

From Figure 3.4, we see that the shape parameter is fundamental for evaluating scheduler performance. We notice that FSPE+PS has *almost optimal performance for all shape values considered* with the default sigma=0.5, while SRPTE and FSPE perform poorly for highly skewed workloads. Regarding non size-based policies, PS is outperformed by LAS for heavy-tailed workloads (shape < 1) and by FIFO for light-tailed ones having shape > 1 ; PS provides a reasonable trade-off when the job size distribution is unknown. When the job size distribution is exponential (shape $= 1$), non size-based scheduling policies perform analogously; this is a result which has been proven analytically (see e.g., the work by Harchol-Balter [27] and the references therein). It is interesting to consider the case of FIFO: in it, jobs are scheduled in series, and the priority between jobs is not correlated with job size: indeed, the MST of FIFO is equivalent to the one of a random scheduler executing jobs in series [32]. FIFO can be therefore seen as the limit case for a size-based scheduler such as FSPE or SRPTE when estimations carry no information at all about job sizes; the fact that errors become less critical as skew diminishes can be therefore explained with the similar patterns observed for FIFO.

Impact of sigma We have seen that the shape of the job size distribution if fundamental in determining the behavior of scheduling algorithms, and that heavy-tailed job size distributions are those in which the behavior of size-based scheduling differs noticeably. Because of this, and since heavy-tailed workloads are central in the literature on scheduling, we focus on those.

In Figure 3.5, we show the impact of the sigma parameter representing error for three heavily skewed workloads. In all three plots, the values for FIFO fall outside of the plot. These plots demonstrate that FSPE+PS is robust with respect to errors in all the three cases we consider, while SRPTE and FSPE suffer as the skew between job sizes grows. In all three cases, FSPE+PS performs better than PS as long as sigma is lower than 2: this corresponds to lax bounds on size estimation quality, requiring a correlation coefficient between job size and its estimate of 0.15 or more.

In all three plots, FSPE+PS performs better than SRPTE; the difference between FSPE+PS and FSPE, instead, becomes discernible only for values of shape < 0.25. We explain this difference by noting that, when several jobs are in the queue, size reduction in the virtual queue of FSPE is slow: this leads to less jobs being late and therefore non preemptable. As the distribution becomes more heavy-tailed, more jobs become late in FSPE and differences between FSPE and FSPE+PS become significant, reaching differences of even around one order of magnitude.

In particular in Figure 3.5(b), there are areas ($0.5 < \text{sigma} < 2$) in which increasing errors decreases (slightly) the MST of FSPE. This counter intuitive phenomenon is explained by the characteristics of the error distribution: the mean of the log-normal distribution grows as sigma grows, therefore the aggregate amount of work for a set of several jobs is more likely to be over-estimated; this reduces the likelihood that several jobs at once become late and therefore non-preempt-able. In other words, FSPE works better with estimation means that tend to over-estimate job size; it is however always better to use FSPE+PS, which provides a more reliable and performing solution to the same problem.

3.1.2 HFSP: Size-Based Scheduling for Hadoop

The promising results described in Section 3.1.1 lead us to implementing HFSP, a size-based scheduler for Hadoop.

Instead of the FSPE+PS strategy suggested in the previous Section, HFSP can deal with estimation errors by leveraging on the fact that the Hadoop framework provides information about job progression: therefore, HFSP starts with a first rough estimation of job size based simply on the number of *map*/*reduce* tasks in a job, but refines this estimation once a few tasks for that job are completed.

This section provides a synthesis of the work that we published in [36]: we refer the interested reader to that publication for more details.

Implementing a size-based scheduling protocol in Hadoop raises a number of challenges. A few of them come from the fact that MapReduce jobs are scheduled at the lower granularity of *tasks*, and that they consist of two separate phases -- *map* and *reduce* -- which are scheduled independently (Sec. 3.1.2.1). In addition, job size is in general unknown a priori: to evaluate it, we develop an *estimation* module (Sec 3.1.2.4) that provides, at first, a coarse estimation of job size upon submission, and then refines it after the first few *sample* tasks have been run. Estimations are used by an *aging* module (Sec. 3.1.2.3) which outputs job priorities; finally the *scheduler* (Sec. 3.1.2.2) uses such priorities to allocate resources while ensuring that sample tasks are allocated quickly, to converge rapidly to a more accurate job size estimation. Next, we describe the components of our system.

3.1.2.1 Hadoop Scheduling

In Hadoop, the *JobTracker* coordinates the worker machines, called *TaskTracker*s. The scheduler resides in the *JobTracker* and allocates *TaskTracker* resources to running tasks: *map* and *reduce* tasks are granted independent *slots* on each machine.

The scheduler is called whenever one or more task slots become free, and it decides which tasks to allocate on those slots.

When a *single* job is submitted to the cluster, the scheduler assigns a number of *map* tasks equal to the number of partitions of the input data. 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 *map* phase may execute in multiple ``waves'', especially when processing very large data. Similarly, *reduce* tasks are scheduled once intermediate data, output from mappers, is available.[4] When *multiple* jobs are submitted to the cluster, the scheduler allocates available task slots across jobs.

In this work we consider the Hadoop Fair Scheduler, which we call FAIR. FAIR groups jobs into ``pools'' (generally corresponding to users or groups of users) 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. When a slot becomes free and needs to be assigned a task, FAIR proceeds as

[4] More precisely, a ``slowstart'' setting indicates the fraction of mappers that are required to finish before reducers are awarded execution slots.

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 resources.

3.1.2.2 The Job Scheduler

In our architecture, the scheduler operates on a set of job priorities that are output by the *aging* module (Sec. 3.1.2.3), which uses job size information provided by the *estimation* module (Sec. 3.1.2.4). Next, we highlight the main issues that we encountered while implementing our scheduler, and we motivate our design choices.

Job Preemption Unlike the abstract protocols shown in Section 3.1.1.1, which schedule full jobs, here scheduling is performed at the *task* granularity. From an abstract point of view, when the priority of a running job is lower than the one of a waiting task, the running job should be *preempted* to free resources for the other. In Hadoop, this can be implemented either by *killing* the running tasks of the preempted job, or by simply *waiting* for those tasks to complete. Note that scheduling choices are more critical in situations of high load, and that the choice of killing running tasks may result in increasing load even more, because the work done by killed tasks should be performed again. As such, we opt here for wait-based preemption. In Section 3.1.3, we show our efforts towards a more efficient preemption primitive.

Job Phases In MapReduce, a job is composed by a *map* phase followed (optionally) by a *reduce* phase. We estimate job size by observing the time needed to compute the first few tasks of each phase; for this reason we cannot estimate the length of the *reduce* phase when scheduling *map* tasks. For the purpose of scheduling choices we consider *map* and *reduce* phases as two separate jobs. For ease of exposition, we thus refer to both *map* and *reduce* phases as ``jobs'' in the remainder of this section. As we experimentally show in Section 3.1.3.3, the good properties of size-based scheduling ensure shorter mean response time for both the *map* and the *reduce* phase, resulting in better response times overall.

Priority to Training Initially, the estimation module provides a rough estimate for the size of new jobs. This estimate is then updated after the first s sample tasks of a job are executed. To guarantee that job size estimates quickly converge to more accurate values, the scheduler gives priority to sample tasks across jobs -- up to a threshold of t% of the total number of slots. Such threshold avoids starvation of ``regular'' jobs in case of a bursty job arrival pattern.

Data locality For performance reasons, it is important to make sure that *map* tasks work on local data. For this reason, we use the *delay scheduling* strategy [50], which postpones scheduling tasks operating on non-local data for a fixed amount of attempts; in those cases, tasks of jobs with lower priority are scheduled instead.

Scheduling Policy As a result of all the choices described above, our scheduling policy -- which is called whenever a task slot frees up -- behaves as follows:

1. Select eligible jobs: those with tasks waiting to be scheduled that conform to the delay scheduling constraints;
2. Sort them according to the priorities obtained from the aging module;
3. Check if sample tasks are running on less than t% of the slots, and if one or more eligible jobs need to execute sample tasks:
   (a) If so, schedule a *sample* task from the highest priority of such jobs;
   (b) Otherwise, schedule a task from the highest priority eligible job.

3.1.2.3 Aging Module

The aging module of HFSP is inspired by the FSPE policy we describe in Section 3.1.1.2. FSPE ``ages'' jobs by reducing the estimated job size by the amount of service they receive in a virtual simulated PS system; analogously, HFSP performs aging according to the amount of service jobs obtain in a simulated FAIR-like scheduler.

The aging module takes as input job size estimates produced by the estimation module, and outputs a priority for each active job, which is used by the scheduling module described above.

To do that, we adopt the notion of *virtual time*, a technique used in many practical implementations of well-known schedulers [34, 22, 26]. Essentially, we keep track of the amount of remaining work for each job in a virtual ``fair'' system, and update it every time the scheduler is called; job priorities are then output sorted by amount of remaining work. While the remaining work does not necessarily reflect accurately the completion time for queued jobs, the *order* in which those jobs complete in virtual time is all that matters for size-based scheduling to work.

Job aging avoids starvation, achieves fairness, and it requires minimal computational load, since the virtual time does not incur in costly updates for jobs already in queue [34, 22].

Max-Min Fairness The estimation module outputs job sizes in a ``serialized form'', that is the sum of runtimes of each task. As such, the physical configuration of the cluster does not influence estimated size. In the virtual time, instead, this becomes a factor: for example, a job requiring only a few tasks cannot occupy the whole virtual cluster, which has the same number of compute slots of the real one. We simulate a *Max-Min Fairness* criterion to take into account jobs that request *less* compute slots than their fair share (*i.e.*, $1/n$-th of the slots if there are n active jobs): a round-robin mechanism allocates virtual cluster slots, starting from small jobs (in terms of the number of tasks). As such, small jobs are implicitly given priority, which reinforces the idea of scheduling small jobs as soon as possible.

Job Aging Each job arrival or task completion triggers a call to the job aging function, which decreases the remaining amount of work for each job according to the virtual allocation described above and to the time that has passed from the last invocation of the aging function. The priorities output by the module correspond to the remaining amount of work per job, so that jobs with the least remaining work in the virtual time will be scheduled first.

Failures The aging module is robust with respect to failures, and supports cluster size upgrades: the max-min fairness allocation uses the information about the number of slots in the system which is provided by the Hadoop framework; once Hadoop detects a failure, job aging will be slower. Conversely, adding nodes will result in faster job aging.

Job Priority and QoS Our scheduler does not currently implement a concept of job priority; however, the aging function can be easily modified to simulate a Generalized Processor Sharing discipline, leading to a scheduling policy analogous to Weighted Fair Queuing [43].

3.1.2.4 Job Size Estimation

Size-based scheduling requires knowledge of job size. In Hadoop, such information is unavailable until a job completes; however, a first rough estimate of job size can use job characteristics known *a priori* such as the number of tasks; after the first sample tasks have executed, the estimation can be updated based on their running time.

The estimation component has been designed to result in minimized response time rather than coming up with perfectly accurate estimates of job length; this is the reason why sample tasks should not be too many

(our default is $s = 5$), and they are scheduled quickly. We stress that the computation performed by the sample tasks is *not* thrown away: the results computed by sample tasks are used to complete a job exactly as those of regular tasks.

Initial Estimation In Hadoop, the number of *map* and *reduce* tasks each job needs is known *a priori*. In turn, each *map* task processes an *input split*: data essentially residing on a single, fixed-size, HDFS block. Our first job size approximation is therefore directly proportional to the number of tasks per job.

The size of a *map* (resp. *reduce*) job with k tasks is, at first, estimated as $\xi \cdot k \cdot l$, where l is the average size of past *map* (resp. *reduce*) tasks, and $\xi \in [1, \infty)$ is a tunable parameter that represents the propensity the system has to schedule jobs of unknown size. At the extreme $\xi = 1$, new jobs are scheduled quite aggressively based on the initial estimate, with the possible drawback of scheduling particularly large jobs too early. More conservative choices of $\xi > 1$ avoid this problem, but might result in increased response times by scheduling jobs later. We note that particularly small jobs, with s or less tasks, are scheduled immediately and finish in the training phase.

***map* Phase Size** It has been observed, across a variety of jobs, that *map* task execution times are generally stable and short [50, 16]. It is thus reasonable to perform job size estimation using only s sample tasks, albeit runtime skew may induce inaccurate size estimation. We recall here that the aging module described above does not require perfect accuracy.

Our estimation uses a measure of the execution time $\sigma_{i,j}$ for each sample task j of job i. For each job, we obtain an estimate of the *map* phase size by multiplying the average (sample) task runtime by k, which is the number of *map* tasks for the estimated job.

Data Locality A *map* sample task could perform worse than normal due to network latencies if operating on non-local data. However, since the sample tasks are between the first to be scheduled, there is a larger choice of blocks to process, making the need of operating on remote data less likely. In combination with the delay scheduling strategy described in Section 3.1.2.2, we found that data locality issues on sample tasks, as a result, are negligible.

***reduce* Phase Size** The *reduce* phase can be broken down in two parts: *shuffle* time -- needed to move and merge data from mappers to reducers -- and the execution time of the *reduce* function, which can only start when the *shuffle* phase has completed.

Size of *shuffle* As soon as a *reduce* task is scheduled, it starts pulling data from the mappers; once data from all mappers is available, a *global sort* is performed by merging all the mappers' output. Since each mapper output is already locally sorted, a linear-time merge step is sufficient.

Thus, an approximate duration of the *shuffle* phase can be computed as follows. *For each* of the s sample *reduce* tasks of a job, we measure the time required for their *shuffle* phase to complete. This is given by the difference between the moment a task executes the *reduce* function, and the moment the same task was scheduled in the training module. The estimated *shuffle* time of the entire *reduce* phase is then the *weighted average* of the individual *shuffle* times of the sample tasks multiplied by the total number of *reduce* tasks of the job, where the weights are the normalized input data size to each sample task.

Execution Time The execution time of the *reduce* phase is evaluated analogously to the *map* phase described before. However, *reduce* tasks can be orders of magnitude longer than *map* tasks, therefore we aim at providing an estimate of the duration of the sample tasks before their completion. In particular, we set a timeout Δ. If a sample task j of job i is not completed by the timeout, its estimated execution time will be $\tilde{\sigma}_{i,j} = \frac{\Delta}{p_{i,j}}$, where $p_{i,j}$ is the *progress* done 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.

Table 3.2: Job sizes in our experimental workloads.

Dataset size	Map tasks	Workload	
		SMALL	LARGE
1 GB	< 5	65%	0%
10 GB	$10 - 50$	20%	10%
40 GB	$50 - 150$	10%	60%
100 GB	> 150	5%	30%

Once we obtain the size (or an estimation of it) for each sample task, we compute the total execution time using the same procedure described in Section 3.1.2.4. The final estimate of the whole *reduce* phase is obtained by adding the estimated *shuffle* time to this estimated execution time.

3.1.2.5 Experiments

This Section focuses on a comparative analysis between the FAIR and HFSP schedulers.

Experimental Setup The cluster is composed by 36 *TaskTracker* machines with 4 CPUs and 8 GB of RAM each. We configured Hadoop according to advised best practises [6, 7]: the HDFS block size is 128 MB, with replication factor of 3; each *TaskTracker* has 2 map slots with 1 GB of RAM dedicated to each and 1 reduce slots with 2 GB of RAM. In total, our cluster has 72 *map* slots and 36 *reduce* slots. The slowstart factor is configured to start the *reduce* phase for a job when 95% of its *map* tasks are completed.

HFSP operates with the following parameters: the sample set size s for both *map* and *reduce* tasks is set to 5; the Δ timeout to estimate *reduce* task size is set to 10 seconds; we schedule aggressively jobs that are in the training phase, setting $\xi = 1$ and $t = 100\%$. The FAIR scheduler has been configured with a single job pool.

Workloads We generate workloads using PigMix [8], a benchmarking suite used to test the performance of Apache Pig releases. PigMix is appealing to us because, much like its standard counterparts for traditional DB systems such as TPC [44], it generates realistic datasets with properties such as data skew, and defines queries inspired by real-world data analysis tasks.

We generated four datasets of sizes respectively 1 GB, 10 GB, 40 GB and 100 GB. Job arrival follows a Poisson process, and jobs are generated by choosing uniformly at random a query between the 17 defined in PigMix, and applying it to one of the datasets according to a workload-defined probability distribution. We evaluate two workloads:

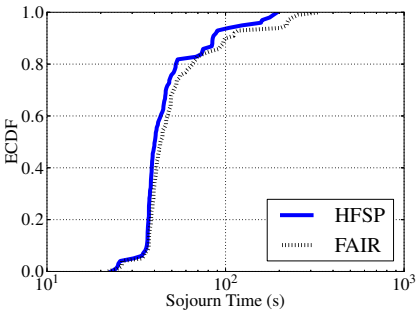
- **SMALL:** this workload is inspired by the Facebook 2009 trace observed by Chen *et al.* [16], where a majority of jobs are very small. The mean interval between job arrivals is $\mu = 30\ s$.
- **LARGE:** this workload is predominantly composed of relatively heavy-duty jobs. In this case, the mean interval between jobs is $\mu = 120\ s$.

In Table 3.2, we report the probability distribution for choosing a particular dataset size; we remark that PigMix queries operate on different subsets of the generated datasets, resulting in a variable number of *map*/*reduce* tasks. Each workload is composed of 100 jobs, and both HFSP and FAIR have been evaluated using the same jobs, the same inputs and the same submission schedule.

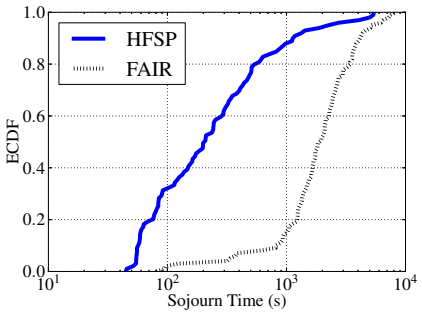
We have additional results -- not included here for conciseness -- that confirm our results on different platforms (Amazon EC2 and the Hadoop Mumak emulator), and with different workloads (synthetic traces generated by SWIM [16]. They are available in a technical report [38].

Table 3.3: Mean sojourn time (MST) and mean load.

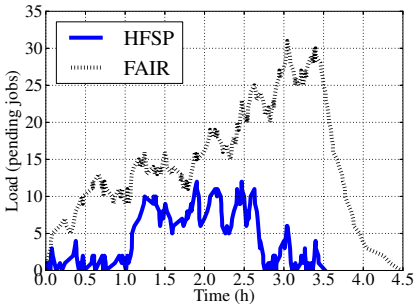
Workload	MST (s)		Mean Load	
	FAIR	HFSP	FAIR	HFSP
SMALL	63	53	2.26	1.99
LARGE	2,291	544	16.80	4.60



(a) Sojourn time for the SMALL workload.



(b) Sojourn time for the LARGE workload.



(c) Cluster load for the LARGE workload.

Figure 3.6: Macro benchmark results.

Macro Benchmarks In order to evaluate the overall performance of our system, we compare FAIR with HFSP on sojourn time -- the interval between a job's submission and its completion -- and load, in terms of number of pending jobs (i.e., those that have been submitted and not yet completed). Table 3.3 shows mean sojourn time (across all jobs) and mean load (over the duration of the experiment) for our two workloads.

In the SMALL workload, HFSP decreases the mean sojourn time by around 16%. By observing the empirical cumulative distribution function (ECDF) of sojourn times in Figure 3.6(a), we notice larger differences between FAIR and HFSP for jobs with longer sojourn times (note the logarithmic scale on the x axis). In this workload, the system is on average loaded with around 2 pending jobs (see Table 3.3); since these jobs are often small, the system is generally able to allocate all tasks of pending jobs, resulting in analogous scheduling choices (and therefore sojourn time) for both FAIR and HFSP. However, when system load is higher, HFSP outperforms FAIR.

Our results are strikingly different for the LARGE workload (Figure 3.6(b)), where the mean sojourn time with HFSP is less than a quarter of the one with FAIR. In this workload, most jobs require several task slots, and complete more quickly since HFSP awards them the entire cluster (if needed) when they are scheduled. Instead, the sharing strategy of FAIR has the drawback of increasing the sojourn time of all jobs. *map* phases of most jobs complete earlier in HFSP, making it possible to schedule *reduce* phases sooner than with FAIR. As a result, with HFSP, 30% of jobs complete within 100 seconds from their submission, while in the same time window FAIR only completes 2% of them; after 1,000 seconds from submission, 90% of jobs are completed with HFSP while only 15% are completed with FAIR.

Scheduling choices are more critical when the cluster is loaded by jobs that require many resources, and the difference between the SMALL and LARGE workloads exemplifies this clearly. Figure 3.6(c) shows the evolution of load run on the LARGE workload: even if the job submission schedule for HFSP and FAIR is the same, load is promptly decreased in HFSP by focusing resources on single jobs.

These results allow us to conclude that HFSP performs better than FAIR in two very different workloads; the advantage is more pronounced when the job and workload size is large with respect to the cluster size. In that case, scheduling decisions become critical, and the inefficiencies of simple fair sharing become apparent.

3.1.3 OS-Assisted Task Preemption for Hadoop

In order to avoid wasting computation, HFSP does not kill tasks of jobs that should have been preempted; instead, it simply waits for them to finish. This choice conserves work, but it is sub-optimal: in this Section, we describe our efforts in creating a new way of handling preemption for Hadoop, namely by suspending and resuming work.

We present here a summary of the work we published in [37]. We refer the interested reader to that work for more detail, also discussing implications on task implementation and on the design of scheduling and eviction strategies..

We now describe our preemption primitive, that implements task suspension and resume operations. First, we outline how process suspension and memory paging work in modern operating systems.

Then, we present the implementation of our preemption mechanism. Note that this work focuses solely on preemption primitives, and glosses over *task eviction policies* that are within the scope of a job and task scheduler.

3.1.3.1 Suspension and Paging in the OS

Here we provide a synthetic description of the way OSes perform memory management, which motivate our design and implementation. A more in-depth description of such mechanisms can be found, for example, in the work of Arpaci-Dusseau [10, Chapters 20 and 21].

In general, system RAM is occupied by file-system (disk) cache and runtime memory allocated by processes (including map/reduce tasks); when RAM is full -- for whatever reason -- the OS needs to *evict* pages from

memory, either by reclaiming space (and evict pages) from the file-system cache or by *paging out* runtime memory to the swap area. Since Hadoop workloads involve large sequential reads from disks, it is a best practice to configure the Linux kernel to give precedence to runtime memory, always evicting file-system cache first [1]. The system therefore only pages out runtime memory to avoid ``out of memory'' conditions, i.e., when the memory allocated by *running* processes exceeds the physical RAM.

To decide which pages to swap to disk, OSes generally employ a policy which is a variant of least-recently-used (LRU) [2]; *clean* pages -- i.e., those that have not been modified since the last time they have been read from disk -- do not need to be written and get prioritized when performing eviction. Page-out operations are generally clustered to improve disk throughput (and amortize on seek costs) by writing multiple pages to disk in a batch. These implementation policies ensure that paging is efficient and with small overheads, especially if a suspended processes leads to swapping. Most importantly for our case, pages from suspended processes are evicted before those from running ones.

We recall that it is necessary to make sure that the aggregate memory size for all processes -- both running and suspended -- does not exceed the size of the swap space on disk, because in such a case the operating system would be forced to kill processes. Since Hadoop tasks can only allocate a limited amount of memory, this can be ensured by configuring the scheduler so that also the number of suspended tasks per task-tracker is limited.

Thrashing. Paging, in general, is not problematic unless *thrashing* happens, a phenomenon where data is continuously read from and written to swap space [20] on disk. Thrashing is caused by a *working set* -- i.e., the set of pages accessed by running programs -- which is larger than main memory.

In Hadoop, thrashing is avoided because two mechanisms are in place: *i)* the number of running tasks per machine is limited (and controlled via a configuration parameter); and *ii)* the heap space size that a given task can allocate is limited (and also controlled via configuration). Proper Hadoop configuration can thus limit working set size and avoid thrashing.

The aforementioned mechanisms prevent thrashing in the same way even when suspension is used. Memory allocated by suspended processes is *outside the working set* and hence *cannot cause thrashing*; pages allocated for the suspended processes are paged out and in *at most once*, respectively after suspension and resuming. Thrashing could only happen if a given job is continuously suspended and resumed by the scheduling mechanism: the moderate cost of a suspend-resume cycle can be thus multiplied by the number of cycles. A reasonable scheduler implementation should take into account that suspending and resuming a job has a cost, and should take measures to avoid paying it too often.

3.1.3.2 Implementation Details

The concepts that we illustrate here are valid for both Hadoop 1 [5], which is the most widely used Hadoop implementation in production, Hadoop 2, which uses a new infrastructure for resource negotiation called YARN [46], and even other frameworks such as Spark [9]. Currently, our implementation targets Hadoop version 1.

Our preemption primitive exposes an API that can be used both by users on the command line and by schedulers. Mirroring the implementation of the `kill` primitive in Hadoop, we introduce *i)* new messages between the JobTracker (a centralized machine responsible for keeping track of system state and scheduling) and TaskTrackers (machines responsible for running Map/Reduce tasks), and *ii)* new identifiers for task states in the JobTracker.

JobTracker Hadoop has a ``heartbeat'' mechanism where, at fixed intervals and every time a task finishes, TaskTrackers inform the JobTracker about their state.

As soon as the JobTracker receives the command to suspend a task from the user or the scheduler, that task is marked as being in a `MUST_SUSPEND` state. At the following heartbeat from the involved TaskTracker, the JobTracker piggybacks the command to suspend the task. The following heartbeat notifies the JobTracker whether the task has been suspended -- which triggers entering the `SUSPENDED` state in the JobTracker -- or

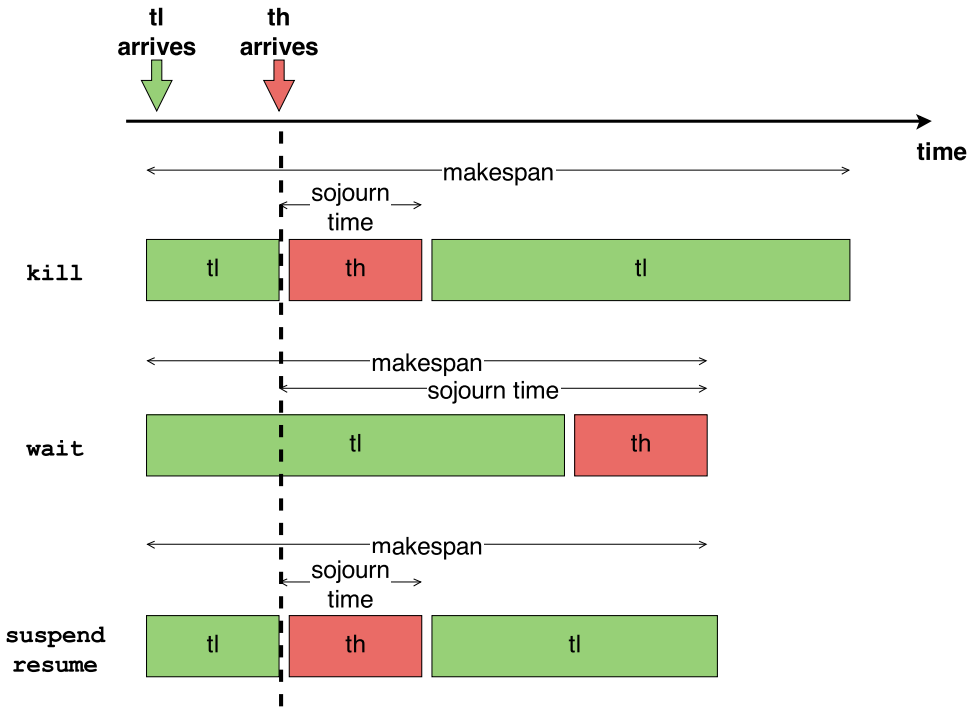


Figure 3.7: Task execution schedules.

whether it completed in the meanwhile.

Analogous steps are taken to resume tasks, exchanging appropriate messages and handling the `MUST_RESUME` state, returning the state to `RUNNING` when the process is over.

TaskTracker In Hadoop, Map and Reduce tasks are regular Unix processes running in child JVMs spawned by the TaskTracker. This means that they can safely be handled with the POSIX signaling infrastructure. In particular, to suspend and resume tasks, our preemption primitive uses the standard POSIX `SIGTSTP` and `SIGCONT` signals.

These signals are used because (unlike `SIGSTOP`) they allow handlers to be written to manage external state, e.g., when closing and reopening network connections.

Job and Task Scheduler We factor out the role of task eviction policies implemented by the scheduler, which are not the focus of this work, by building a new scheduling component for Hadoop -- a dummy scheduler -- which dictates task eviction according to static configuration files. This allows to specify, using a series of simple triggers, which jobs/tasks are run in the cluster and which are preempted. In addition to executing jobs and preempting tasks with our `suspend/resume` primitives, the dummy scheduler also allows using the `kill` primitive and to `wait`, for the purpose of a comparative analysis.

3.1.3.3 Experimental Evaluation

In our experiments, we evaluate preemption primitives in terms of the latency they introduce and the amount of redundant work they require. We show that our approach outperforms other preemption primitives and has a small overhead both when jobs are lightweight in terms of memory, and when they are memory-hungry.

Experimental Setup Our `suspend/resume` primitives operate at the task level, and behave in the same way for both Map and Reduce tasks. We evaluate the behavior of the system in a simple setup: our dummy scheduler runs two single-task, map-only jobs, called t_h and t_l (h and l stand for high and low priority respectively). t_l processes a single-block file stored on HDFS, with size 512 MB; t_h processes single HDFS input block of size 512 MB. Both jobs run synthetic mappers, which read and parse the randomly generated input. We remark that this setup is analogous to the one used by Cho et al., who evaluated their preemption primitive using similar synthetic jobs created by the SWIM workload generator [15].

In our experiments, our dummy scheduler preempts the low-priority task t_l after it has reached a completion rate r% (i.e., r% of the input tuples have been processed) and grants the task slot to the high priority task t_h. Once t_h is completed, the scheduler resumes t_l, which can complete as well.

Next, we evaluate the behavior of our `suspend/resume` preemption mechanism against the two baseline primitives available in Hadoop: `wait` and `kill`. When waiting, task t_h is simply executed after t_l completes; when killing, task t_l is killed as soon as t_h is scheduled, and t_l is rescheduled from scratch after the completion of t_h. This simple experimental setup is illustrated in Figure 3.7.

According to Hadoop configuration best practices, in our experimental setup we prioritize runtime memory over disk cache and therefore limit swapping, as discussed in Section 3.1.3.1, by setting the Linux `swappiness` parameter to 0.

Performance Metrics Our goals are ensuring low latency for high-priority tasks, and avoid wasting work: we quantify them, respectively, with the *sojourn time of* t_h and the *makespan* of the workload. *Sojourn Time of* t_h is the time that elapses between the moment t_h is submitted and when it completes; *makespan* is the time that passes between the moment in which the first task t_l is submitted and when *both* tasks are complete.

Results We focus on experimental results in case of light-weight tasks. This is the standard case for ``functional'', stateless, mappers and reducers. In this case, the amount of memory that tasks allocate is essentially due to the Hadoop execution engine (i.e., JVM, I/O buffers, overhead due to sorting, etc.).

Stateful mappers and reducers, instead, can allocate non-negligible amounts of memory; we thus complement our experiments by studying our performance metrics and overheads for memory-hungry jobs, which represent a worst-case scenario for our preemption primitive.

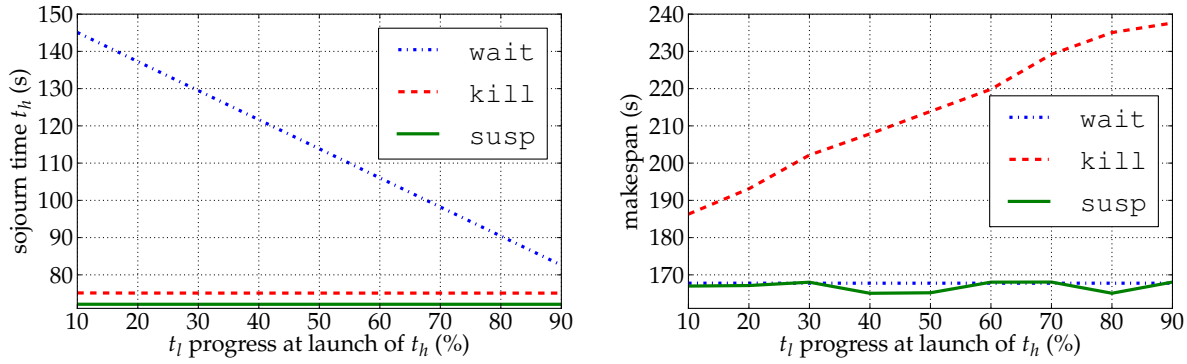
All our results are obtained by averaging 20 experiment runs; we omit error bars for readability: in all data points reported, minimum and maximum values measured are within 5% of the average values.

Baseline Experiments Figure 3.8(a) illustrates the sojourn time of t_h: the arrival rate of $_h$ is a parameter defined as a function of t_l progress, as shown on the x-axis.

The `kill` and our `suspend/resume` primitives achieve small sojourn times, as opposed to `wait`, in particular when t_h arrives early. However, they both incur in some overheads: `kill` runs a cleanup task to remove temporary outputs of the killed task; `suspend/resume` may slow down t_h in case paging out memory occupied by t_l is needed. In our baseline setup, both jobs are light-weight, hence the suspended process resides only in memory. This explains the small advantage for our mechanism, which outperforms all other primitives even when t_h arrives at 90% completion rate of task t_l.

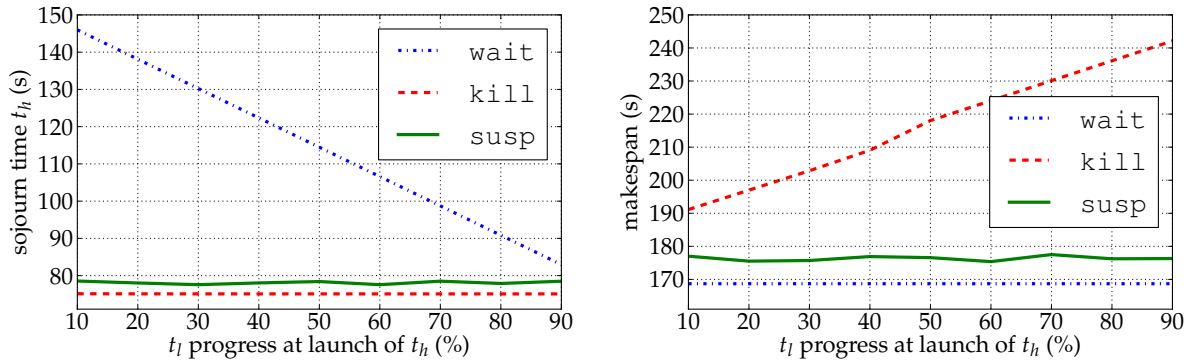
Figure 3.8(b) illustrates our results for the makespan metric, using the same setup described above. In this case, the makespan is heavily affected by a preemption primitive that wastes work. The `wait` policy, at the cost of delaying t_h, avoids supplementary work and achieves a small makespan; the `kill` primitive, instead, wastes all the work done by t_l before preemption. Finally, our preemption primitive behaves similarly to the `wait` policy, despite the possible overhead due to page-out/page-in cycles.

For light-weight jobs, we conclude that our primitive is superior to both alternatives, as both sojourn times and makespan are small. We note that the authors of Natjam measured an overhead of around 7% in terms



(a) Sojourn time of t_h (b) Makespan

Figure 3.8: Baseline experiments: a comparison of the three preemption primitives with light-weight tasks.



(a) Sojourn time of t_h (b) Makespan

Figure 3.9: Worst-case experiments: a comparison of the three preemption primitives with memory-hungry tasks.

of makespan, in similar experimental settings as ours. Our findings suggest that the overhead in our case is negligible.

Worst-Case Experiments The experiments discussed above are valid for simple implementations of Map and Reduce tasks, that carry out stateless computations on their input. Stateful tasks can, however, allocate memory, which may force the OS to swap. Since clusters often have plentiful available memory [4], such a situation is unlikely to be frequent. However, we still consider a ``worst case'' scenario to stress our primitive: both t_l and t_h allocate a large amount of memory (2 GB in our case; we note that this requires an *ad hoc* change to the Hadoop configuration since Hadoop jobs are not generally allowed to allocate such an amount of memory). This value makes sure that, when running a single task the system does not have to recur to swap;[5] conversely, when the two tasks are present in the system at the same time, one of them is forced to page out memory. We ensure that tasks allocate memory and that the OS marks pages as ``dirty'', by writing random values to all memory at task startup, and reading them back when finalizing the tasks.

Figures 3.9(a) and 3.9(b) present the sojourn time and the makespan for the worst-case experimental setup. While our preemption primitive still outperforms both alternatives with respect to both metrics, it is possible to notice that the overheads related to paging are visible: with respect to the sojourn time, the `kill` primitive

[5]The physical memory of our system is 4 GB; the rest of the memory is needed by the Hadoop framework and by the operating system services.

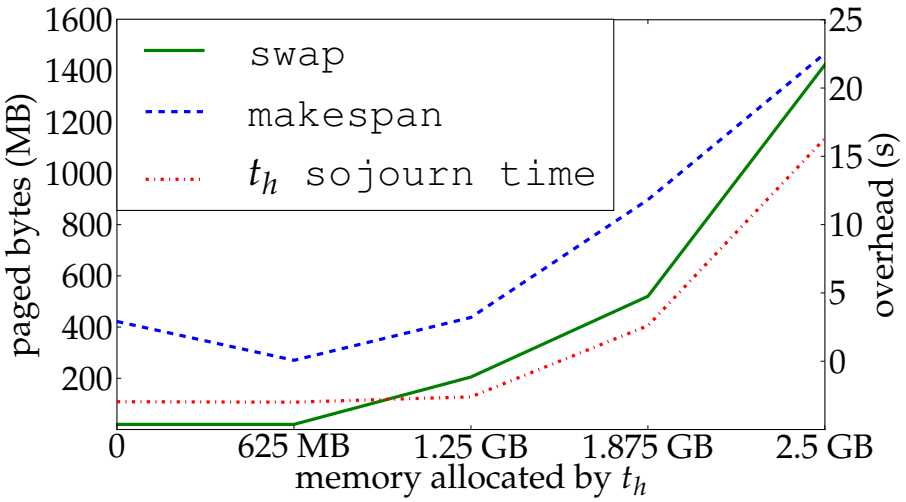


Figure 3.10: Overheads when varying memory usage.

achieves a slightly lower value; similarly, the `wait` primitive achieves slightly smaller makespan. Overall, the overhead due to our preemption primitive is marginal: we further investigate and quantify it in the next section.

Impact of Memory Footprint We now focus on a detailed analysis of the overheads imposed by the OS paging mechanism on the performance of our preemption primitive. To do so, we vary the amount of memory a task allocates in the setup phase.[6] In our experiments t_l allocates 2.5 GB of memory, and we parametrize over the amount of memory t_h allocates. For each experimental run, we measure the number of bytes swapped by the process executing t_l, and compute the degradation of sojourn time and makespan compared to the `kill` and `wait` primitives, respectively.

Figure 3.10 indicates that the overheads due to paging are roughly linearly correlated to the amount of data swapped to disk. For the sojourn time, our preemption primitive degrades when t_h allocates more than 1.5 GB of RAM: in the worst-case, sojourn time is 20% larger than with the `kill` primitive. Similarly, for the makespan, our mechanism degrades when t_h allocates more than 1.3 GB: in the worst-case, makespan is 12% larger than with the `wait` primitive. Finally, we note that swapped data grows more than linearly because of an approximate implementation of the page replacement algorithm in Linux (and other modern OSes), which can lead to more swapping than strictly necessary [14, Chapter 17].

3.2 `Schedule`: Cache-Oblivious Scheduling of Shared Workloads

Shared workload optimization is feasible if the set of tasks to be executed is known in advance, as is the case in updating a set of materialized views or executing an extract-transform-load workflow. In this section, we consider data-intensive shared workloads with precedence constraints arising from data dependencies, i.e., before executing some task, other tasks may have to run first and generate some data needed by the next task(s). While there has been previous work on identifying common sub-expressions in shared workloads and task re-ordering to enable shared scans, in this section we go a step further and solve the problem of scheduling shared data-intensive workloads in a cache-oblivious way.

Our solution relies on a novel formulation of precedence constrained scheduling with the additional constraint that once a data item is in the cache, all tasks that require this data item should execute as soon as possible thereafter. The intuition behind this formulation is that the longer a data item remains in the cache,

[6] This is where, generally, auxiliary data structures are created to maintain an internal state in a task.

the more likely it is to be evicted regardless of the cache size. We give an optimal ordering algorithm using A* search over the space of possible orderings, and we propose efficient and effective heuristics that obtain nearly-optimal results in much less time.

3.2.1 Introduction to the Problem

There are several data management scenarios in which the workload consists of concurrent tasks that are known in advance. For example, extract-transform-load (ETL) processing involves executing a predefined workflow of operations that pre-process data before inserting it into the database. Another example is data stream processing and publish-subscribe systems, in which a predefined set of queries is continuously executed on incoming data. Also, in data warehouses, materialized view maintenance is often done periodically, in which all the views, which are known in advance, are updated together.

Previous work has recognized optimization opportunities in these scenarios, referred to as shared workloads, including scan sharing, shared query plans and evaluating common sub-expressions only once [30]. In this work, we go a step further and address the following problem: given a shared workload, even after identifying common sub-expressions and shared scan opportunities, it is still not clear *what is an optimal ordering of operations that maximizes the re-use of cached results?* Furthermore, since we may not know the exact amount of cache that is available to the data management system at a given time, we want to generate a task ordering in a *cache-oblivious way*, i.e., in a way that exploits caching without knowing the cache size.

Throughout this work, we will use the term ``cached results'' in a general sense. Depending on the application, this could refer to the disk-RAM hierarchy or the RAM-cache hierarchy.

Motivating Example

While the solution presented in this work is applicable to any data-intensive shared workload (i.e., where data I/O is the bottleneck, not CPU cycles), our motivation for studying cache-oblivious task ordering comes from Data Stream Warehouses (DSWs) such as DataDepot [25] and DBStream [11]. DSWs are a combination of traditional data warehouse systems and stream engines. They support very large fact tables, materialized view hierarchies and complex analytics; however, in contrast to traditional data warehouses that are usually updated once a day or once a week, DSWs are refreshed more often (e.g, every 5 minutes) to enable queries over nearly-real time and historical data. Example applications include network, datacenter or infrastructure monitoring, data analysis for intelligent transportation systems and smart grid monitoring.

Since the ``claim to fame'' of DSW systems is their ability to ingest new data and refresh materialized views frequently, the view maintenance operations must be performed efficiently. The system must finish propagating one batch of new data throughout the view hierarchy before the next batch arrives. Otherwise, at best a backlog of buffered data will build up, and at worst some data will be lost and not available for future analysis.

Based on our experience with building and tuning DSW systems for network monitoring data, the efficiency of view maintenance operations depends on the order in which they are performed, which affects the extent to which cached data are re-used. For example, consider the simple view hierarchy shown in Fig. 3.11, with 0 and 1 being base tables and the other nodes corresponding to materialized views (real applications can easily have hundreds of views). Note that some views (e.g., 2 and 4) are computed directly over base tables while others are computed over other views (e.g., 5). This is the predefined workload that a DSW will repeatedly execute when a batch of new data arrives for tables 0 and 1.

The view hierarchy illustrated in Fig. 3.11 forms a precedence graph. When a batch of new data arrives for table 1, we must first insert it into table 1, and then we can use it to update views 2 and 4. Since view 5 needs view 2 as an input, it can only start processing after view 2 was updated. Thus, a legal ordering of the view updates must satisfy the given precedence constraints; e.g., we cannot update view 5 if we have not yet updated view 2.

However, different legal orderings may lead to different extents of cache re-use. For example, right after

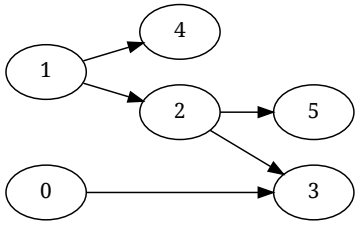


Figure 3.11: A precedence graph corresponding to two base tables and four materialized views.

updating table 1 with new data, that new batch of data is likely to be in the cache. Therefore, we should then update views 2 and 4 while the new batch of data is in the cache. On the other hand, if we update table 0 in between views 2 and 4, then the new batch of data from table 1 is more likely to be evicted and will have to be reloaded before updating view 4. Put another way, with this ordering, we would need a larger cache to avoid cache misses.

Challenges

Even in the simple example above, it is not obvious which ordering makes the best use of cached results. It makes sense to update views 2 and 4 immediately after updating table 1, but should we update view 2 before 4 or vice versa? As the number of tasks in the workload and their data dependencies increase, so does the complexity of choosing an efficient ordering. Furthermore, in practice we usually do not know exactly how much cache is available for a given task at a given time. For instance, in a DSW, view updates compete for resources with ad-hoc queries.

The intuition behind our solution is simple. The longer a data item remains in the cache, the more likely it is to get evicted. Thus, tasks that require some data item should be scheduled as soon as possible after this data item is placed in the cache. In other words, we need to minimize the amount of time a data item (e.g., a new batch of data loaded into a materialized view) spends in the cache until all the subsequent tasks that need it have been executed. Note that these objectives are cache-oblivious in the sense that we do not need to know the cache size to achieve them.

For example, Fig. 3.12 illustrates two possible legal orderings of the five tasks from Fig. 3.11, obtained by linearizing the view precedence graph. For each node that includes at least one outgoing edge (e.g., each task that produces data required by other task(s)), we can compute how long these data must remain in the cache. At the top of the figure, the ``distance'' between table 0 and view 3, which requires data from table 0, is four, i.e., three other tasks will run in between. For view 2, the maximum distance is three, since both view 3 and view 5 need data from view 2, and a total of three view updates will run from the time view 2 data are inserted into the cache until both view 3 and view 5 updates are completed. On the other hand, the ordering shown at the bottom of the figure has a distance of only one between table 0 and view 3---they are executed one after the other and data from table 0 is more likely to still be in the cache at the time of execution of view 3. The idea behind our approach is to minimize the distance between related tasks and therefore increase the likelihood of re-using cached results, without having to know the cache size (our notion of distance will be formalized in Section 3.2.2).

3.2.2 Problem Statement

The general problem we investigate in this work is the scheduling of tasks with precedence constraints corresponding to data dependencies among the tasks. Precedence constraints impose a partial order on the tasks. This partial order is given as input in the form of a directed acyclic graph (DAG) $G = (V, E)$, where each node $v \in V$ represent a task and each directed edge $e = (u, v) \in E$ is a precedence constraint, which requires that task u has to be scheduled before task v. Optionally, the input may include the size the output

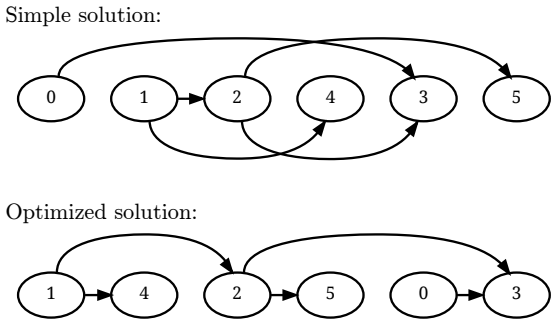


Figure 3.12: A simple and a optimized ordering of the tasks from Figure 3.11.

of each task; we will deal with this later on in this section. In addition to satisfying the given precedence constraints, we will impose optimization goals on the generated ordering to maximize the re-use of cached data without knowing the cache size.

G may consist of a number of connected components, e.g., a view hierarchy that uses some set of base tables, and another view hierarchy that is sourced from a different set of base tables. Since inter-dependencies and therefore cache optimization opportunities only exist in each connected component, we can deal with each connected component separately, and thus we assume from now on that G is connected.

We assume the tasks are data-intensive. That is, the bottleneck is loading the data into the cache rather than the subsequent processing; otherwise, even having an unlimited cache would not help much. We assume a *cache-oblivious* setting, in which we do not know the size or granularity of the cache. Further, we assume that the tasks belonging to a given connected component are to be scheduled serially on a single machine, although different connected components can be scheduled in parallel. We defer a full treatment of multi-threaded scheduling in our context, as well as handling task priorities, to future work.

Let $\sigma : V \rightarrow \{0, 1, ..., |V|\}$ be a schedule function that orders the tasks (i.e., the nodes in the precedence graph) in a given workload. The *precedence constrained scheduling* problem as formulated in [23] asks whether a schedule can meet a deadline. On a single-processor system, the problem of scheduling tasks with precedence constrains, without taking caching into account, is solvable in polynomial time [23]. However, real systems benefit from caching: the result of a preceding task u can be retrieved from the cache for task v, if the cache has enough capacity to keep the result of v despite other tasks that are scheduled between u and v. Therefore, minimizing the distance between u and v in the schedule σ, which is expressed by $|\sigma(u) - \sigma(v)|$, increases the likelihood of a cache hit.

There are two classical problems that express related objectives: (1) directed bandwidth, which aims to construct a schedule with a bound on the maximum distance of an edge in the precedence graph, and (2) directed optimal linear arrangement, which aims to construct a schedule with a bound on the sum of the distances for all edges:

Directed bandwidth (DBW) (GT41 in [23], GT43 in [18]): Given a graph $G = (V, E)$ and a positive integer K, is there a schedule function $\sigma : V \rightarrow \{1, ..., |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\max |\sigma(v) - \sigma(u)| \leq K\,? \tag{3.2}$$

Directed optimal linear arrangement (DLA) (GT42 in [23], cf. GT44 in [18]): Given a graph $G = (V, E)$ and a positive integer K, is there a schedule function $\sigma : V \rightarrow \{1, ..., |V|\}$ such that $\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\sum_{(u,v)\in E} |\sigma(v) - \sigma(u)| \leq K\,? \tag{3.3}$$

Both of the above problems are NP-complete [24, 35]. Note that the problems are defined as decision problems, for which the corresponding optimization problems can be shown to be equally complex.

In our context, solving the DBW problem only optimizes for the single longest edge in the entire workload and does not take any other data dependencies into account. Also, DLA is not suitable in the context of caching as it was originally meant for scheduling of production workloads, in which a task produces multiple items, one for each subsequent tasks it is connected to. For example, recall Fig. 3.12 and note the edges from task 2 to tasks 3 and 5. DLA counts both of these edges, effectively assuming that two copies of the output of task 2 need to be stored. However, *we are only interested in the longest edge from a task to any subsequent task that depends on it*, as that determines how long the data generated by the initial task need to stay in the cache. This way, the resulting schedule is optimized for re-use of cached results irrespective of the cache size.

Based on the above observation, we formulate a new problem, *total maximum bandwidth*, that reflects our objective, which is a combination of DBW and DLA:

Total Maximum Bandwidth (TMB): Given a graph $G = (V, E)$ and a positive integer K, is there a schedule function $\sigma : V \rightarrow \{1, ..., |V|\}$ such that

$\forall (u, v) \in E : \sigma(u) < \sigma(v)$ and

$$\sum_{u \in V} \max_{\{v|(u,v)\in E\}} |\sigma(v) - \sigma(u)| \leq K\,? \tag{3.4}$$

If the input also includes the size of the output of each task u, call it ω_u, then we can extend TMB to *Weighted Total Maximum Bandwidth* (WTMB) by optimizing for the weighted distance of the longest edge from any task to a dependent task. For WTMB, the optimization problem becomes:

$$\sum_{u \in V} \omega_u \max_{\{v|(u,v)\in E\}} |\sigma(v) - \sigma(u)| \leq K\,? \tag{3.5}$$

We close this section with an example of TMB and DBW. Fig. 3.13 (top) shows a precedence graph for five tasks, followed by two possible schedules (A and B). Schedule A) is optimized for TMB and has a TMB score of six: the distance between task 0 and 2 (of 1) plus the distance between task 2 and 4 (of 3), plus the distance between task 1 and 3 (of 1), plus the distance between task 3 and 4 (of 1). Its DBW score is three (the maximum over these distances). Schedule B) is optimized for DBW; its DBW score is two and its TMB score is seven.

Fig 3.13 also shows the data outputs that must be cached throughout the execution of the schedules, assuming an optimal eviction policy. In schedule A), we start with the output of task zero, which can be evicted as soon as task 2 is done. After task 2 is done, its output is in the cache, and the output of task 1 is added when task 1 is done. After task 3 is done, the output of task 1 can be evicted. In schedule B), we also start with the output of task zero, and adding the output of task 1 after it is done, and so on. Notice that schedule A), which optimizes for TMB, requires less cache over time since only one item needs to be stored in the second step, compared to two items in schedule B).

3.2.3 Algorithms

In this section, we present algorithms that take in a precedence graph and output a schedule optimized for TMB or WTBM if the task output sizes are known (Equations 3.4 and 3.5, respectively). We start by defining some of the concepts and subroutines that will be used by the algorithms (Section 3.2.3). We then present an optimal algorithm based on A*-search of the complete space of possible schedules (Section 3.2.3), followed by three approximate algorithms that examine a subset of possible schedules: a simple breadth-first baseline approach (Section 3.2.3) and two heuristics, a greedy algorithm that always chooses a task whose distance to its predecessor is the smallest (Section 3.2.3) and an algorithm that chooses tasks which are likely to lead to efficient schedules (Section 3.2.3).

Preliminaries

First, we define the *candidate search graph*, $\overline{G} = (\overline{V}, \overline{E})$, in which the sequence of edge labels along every path from the start to the sink is a feasible schedule that obeys the precedence constraints encoded by the

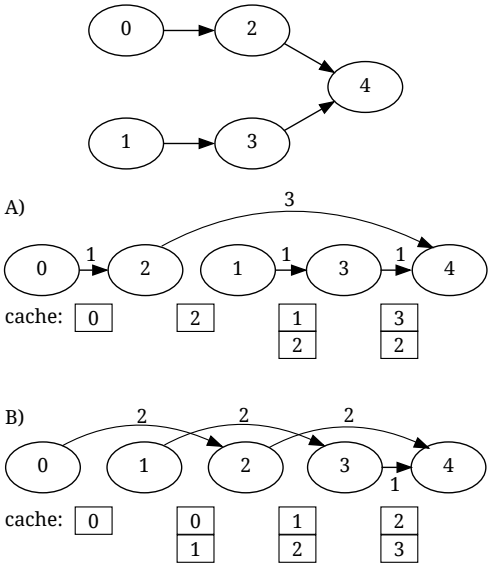


Figure 3.13: An example comparing TMB with DBW.

given precedence graph $G = (V, E)$[7]. For example, the candidate search graph corresponding to the precedence graph from Fig. 3.11 is shown in Fig. 3.14. Each node $\overline{v} \in \overline{V}$ denotes the schedulable tasks at that point in the schedule, i.e., those which can now be executed because all of their precedence constraints have been met. Each edge $(\overline{u}, \overline{v}) \in \overline{E}$ is labeled with the name of the task that is to be executed at that step. The start node at the top of Fig. 3.14 contains tasks 0 and 1, which must run before any other tasks. If we follow the right edge, labeled 1, the schedulable tasks are now 0, 2 and 4, and so on.

We can construct $\overline{G}$ from G in a straightforward way. In the first step, the source node in $\overline{G}$ contains all the root nodes in G (i.e., the tasks without any predecessors). In each subsequent step, we create edges for all tasks contained in the labels of nodes created in the previous step, labeled with the task number. For each of the created edges, we created a new node in $\overline{G}$ that contains the tasks that are now schedulable (if such a node has not already been created). Finally, if no more nodes are schedulable, the edges are connected to the sink node, labeled with $\emptyset$.

Since we create $\overline{G}$ on-the-fly, the following definitions are based on a partial schedule. Let s be a possibly partial schedule of $|s|$ tasks. Let $get\_cands(G, s) := \{v : (u, v) \in E \text{ and } u \in s \text{ and } v \notin s\}$. That is, get_cands returns the set of schedulable tasks that can now be appended to s assuming that all the tasks in s have already been executed. Furthermore, for a task u, let $successors(u)$ be the set of tasks that depend on u, i.e., $successors(u) = \{v : (u, v) \in E(G)\}$.

Finally, we define a $tmb\_cost(s, G)$ function that evaluates the Total Maximum Bandwidth cost of a possibly partial schedule s according to Equation 3.4 (or Equation 3.5 if the task output sizes $\omega(u)$ are known). Let σ be the ordering function of s (recall Section 3.2.2). For each task u in s, we compute $tmb\_cost(s, G)$ as follows.

1. if u has no successors, do nothing
2. else if all of u's successors are already in s, add to the total cost the distance between u's last successor and u, i.e., $\max_{v \in successors(u)} |\sigma(v) - \sigma(u)|$, multiplied by $\omega(u)$ if given
3. else (if not all of u's successors are in s), add to the total cost the quantity $|s| + 1 - \sigma(u)$, which is a lower bound on the distance between u's last successor (which has not yet been scheduled) and u (again, multiplied by $\omega(u)$ if given)

[7] A similar search graph was used in [45] in the context of the direct optimal linear arrangement problem.

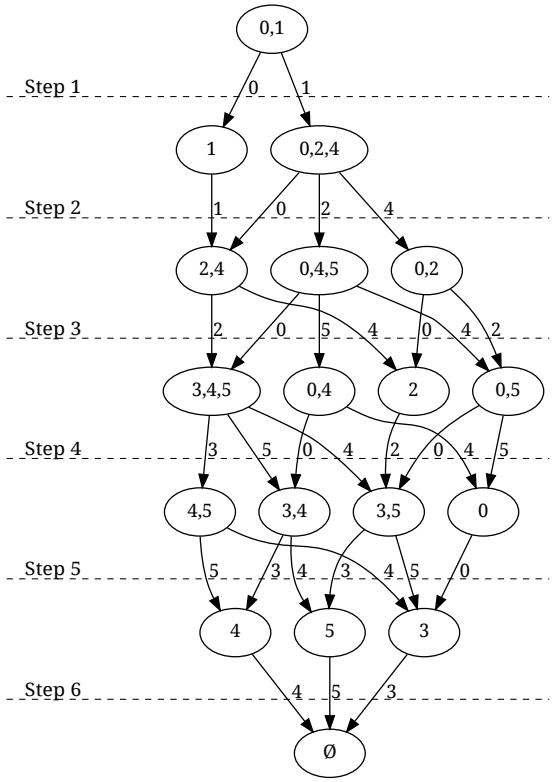


Figure 3.14: Candidate search graph for the workload whose precedence graph was shown in Figure 3.11.

For example, consider the partial schedule $s = \langle 0, 1, 2, 4 \rangle$ for the precedence graph from Fig. 3.11. For task zero, the cost is two since its successor, task 2, appears two positions later in the sequence. For task 1, the cost is 3, which is a lower bound on its true cost (its successor, task 3, has not yet been scheduled). The cost for task 2 is one and the cost for task 4 is one, which, again, is a lower bound on its true cost. Thus, $tmb\_cost(s, G) = 2 + 3 + 1 + 1 = 7$.

Algorithm 2 shows the pseudo-code for tmb_cost. Each task in the possibly partial schedule s is considered sequentially. Lines 6 through 11 count how many of the current task's successors are in s and record the position in s of the furthest successor of the current task. Note the use of the schedule function σ to find the position of outTask in s. Line 12 counts the total number of successors of a given task. If this number is zero, the current task does not incur any TMB cost. Otherwise, if all the successors have already been scheduled, we can precisely compute the TMB cost in line 16, which is simply the difference in the position of the furthest successor and the task itself. Otherwise, line 18 computes a lower bound on the given task's TMB cost. In line 20, we add the cost of the current task to the total cost of the schedule. Note the ω function that determines the output sizes for WTMB. In case of TMB, $\omega(task)$ is simply one for every task.

Optimal Algorithm Based on A* Search

We begin with an optimal algorithm based on A* search that considers every possible schedule and selects an optimal one with the lowest tmb_cost. As we will experimentally show in Section 3.1.3.3, this algorithm is not feasible in practice for non-trivial problem instances because the number of possible schedules can be very large.

A* search finds a least-cost path between two nodes, in our case the start node and the sink node of the candidate search graph (i.e., a least-cost schedule). For each node x, the cost function used by A* includes

Algorithm 2 (Weighted) tmb_cost

```
1:  cost = 0  // the overall cost of the schedule
2:  for task in s do
3:      stepCost = 0
4:      maxOutPos = 0 // position of furthest successor
5:      outTasksDone = 0
6:      for outTask in successors(task) do
7:          if outTask in s then
8:              outTasksDone++
9:              maxOutPos = max(σ[outTask], maxOutPos)
10:         end if
11:     end for
12:     ℓ = len(successors(task))
13:     if ℓ == 0 then
14:         do nothing // no successors
15:     else if outTasksDone == ℓ then
16:         stepCost = maxOutPos - σ(task)
17:     else
18:         stepCost = len(s) - σ(task)
19:     end if
20:     cost += ω(task) * stepCost
21: end for
22: return cost
```

two parts: $g(x)$, which is the cost of the path from the start node to x, and $h(x)$ which is a heuristic function that approximates, but must not overestimate, the cost of the path from x to the sink node. In our problem, $g(x)$ is simply $tmb\_cost(s)$ where s is the schedule corresponding to the path from the start node to x. The more interesting part is $h(x)$.

To solve our problem, we define $h(x)$ as the sum of the outgoing edges present in the precedence graph for each task that has not yet been scheduled along the path from the start node to x. To understand why this is an admissible function for A* search (i.e., one that does not overestimate the remaining cost of the path), note that if a task node has an outgoing edge in the precedence graph, then there is a successor task that must be scheduled after that node. Thus, a lower bound on the total maximum bandwidth cost for the given task is the number of its outgoing edges in the precedence graph, i.e., the number of its successors. This lower bound occurs if all the successors are scheduled immediately after the given task. If any other task is scheduled before the last successor, the cost can only increase.

For example, consider the partial schedule $s = \langle 0, 1, 4 \rangle$ based on Fig. 3.11. The $g(x)$ function of the node in the candidate search graph corresponding to this partial schedule is simply $tmb\_cost(s, G)$, which is 6 (three each for task zero and one, since not all of their successors have been scheduled, and zero for task 4 because it does not have any successors). To compute $h(x)$, note that tasks 2, 3 and 5 are yet to be scheduled. The sum of the outgoing edges of these three nodes in the given precedence graph is two, which gives us $h(x)$. Thus, the total cost of s as computed by A* search is $g(x) + h(x) = 8$. It is easy to verify that no complete schedule with s as its prefix can have a tmb_cost of less than 8.

Baseline Algorithm

We now present the first of three algorithms that consider a subset of the possible schedules and therefore are faster than the A*-based algorithm, but are not guaranteed to find a good solution. The first such algorithm is the simplest and fastest approach we refer to as *Baseline*: at every step, it randomly chooses one of

the currently-schedulable tasks. Thus, using the precedence graph from Fig. 3.11 as input, in the first step, Baseline executes tasks 0 and 1 in random order, then tasks 2, 3 and 4 in random order, and then task 5. The running time of Baseline corresponds to that of breadth-first-search, which is $\mathcal{O}(|V| + |E|)$.

Greedy Algorithm

The next algorithm is the standard greedy heuristic applied to our problem: at every step, it chooses a schedulable task that yields the lowest $tmb\_cost(s, G)$ when added to the current partial schedule s. Ties are broken randomly.

Using the precedence graph from Fig. 3.11 as input, the greedy heuristic first decides between tasks zero and 1. For both s=$\langle 0 \rangle$ and s=$\langle 1 \rangle$, $tmb\_cost(s, G) = 1$ since not all of 0's or 1's successors, respectively, have been scheduled. Suppose the tie-break results in task 1 being sequenced first. In the next step, the schedulable tasks are still zero, plus 2 and 4. For s=$\langle 1, 0 \rangle$, $tmb\_cost(s, G) = 2$. For s=$\langle 1, 2 \rangle$, $tmb\_cost(s, G)$ is 2 due to task 1 plus 1 due to task 2, which gives 3. For s=$\langle 1, 4 \rangle$, $tmb\_cost(s, G) = 2$ due to task 1 (plus zero due to task 4 since it has no successors). Thus, the greedy algorithm randomly chooses between task 0 and 4 to follow task 1. We omit the remaining steps for brevity.

We now analyze the runtime complexity of the greedy algorithm. It uses the get_cands function to retrieve the set of currently schedulable tasks. However, since each step of the algorithm adds one task to the schedule, only the successors of this new task need to be added to the schedulable set. This gives $\mathcal{O}(|V| + |E|)$ for all get_cands calls over all the iterations.

The runtime is dominated by calling tmb_cost for the considered schedules, which requires looping over all the outgoing edges of the tasks already in the schedule. This gives $\mathcal{O}(|E|)$ per call.

Since the algorithm iterates $|V|$ times, and, clearly, at every iteration there are no more than $|V|$ schedulable tasks, for which tmb_cost is evaluated, the overall complexity of the greedy algorithm is $\mathcal{O}(|V|^2|E|)$.

Heuristic Algorithm

Our final algorithm is called *Heuristic*. In contrast to the greedy algorithm, which only examines the tmb_cost of adding every schedulable task to the current schedule in each iteration, the heuristic algorithm computes a complete feasible schedule for each schedulable task in every iteration, and chooses the task with the lowest-cost complete schedule. However, to keep the running time manageable, the heuristic algorithm cannot explore every possible feasible schedule (as does the A* algorithm). Instead, the complete schedules for each schedulable task are heuristically computed via deepest-first traversal, as explained below.

First, the heuristic algorithm pre-processes the precedence graph G by adding depth information to each node, corresponding to the distance to the furthest ancestor. For instance, in Fig. 3.11, the depth of tasks zero and 1 is zero, the depth of tasks 2 and 4 is one, the depth of task 5 is two, and the depth of task 3 is also two (its distance to task zero is one, but the distance to its other ancestor, task 1, is two).

Next, we illustrate what happens in the first iteration using Fig. 3.11 as input. Initially, the only schedulable tasks are zero and 1. We need to build complete schedules starting with zero and 1, respectively, compute their tmb_cost, and choose the task whose complete schedule has a lower tmb_cost (and we break ties arbitrarily).

The complete depth-first schedule that starts with task 1 is computed as follows. After task 1 has been scheduled, the schedulable tasks are zero, 2 and 4, of which either 2 or 4 have the largest depth. Let us assume task 2 is chosen next. The schedulable tasks then become zero, 4 and 5, of which is chosen because its depth is the largest. With the partial schedule now $\langle 1, 2, 5 \rangle$, the schedulable tasks are zero and 4. We choose 4, and finally zero and three. This gives the complete schedule $s = \langle 1, 2, 5, 4, 0, 3 \rangle$. Its tmb_cost is three for task 1, 4 for task 2, and one for task zero, which gives 8.

Similarly, the complete depth-first schedule that starts with task zero is computed as follows. After task zero has been scheduled, the only schedulable task is 1, so we choose it. Next, we have a choice between tasks 2

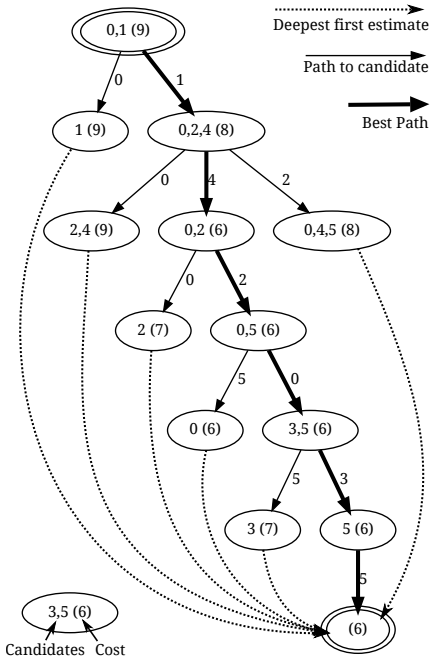


Figure 3.15: Visualization of a run of the Heuristic algorithm on the candidate search graph created from the precedence DAG in Fig. 3.11.

and 4, both of which have the same depth, so let us say we choose task 2. Then, the schedulable tasks are 3, 4 and 5, of which 3 and 5 have the highest depth, so let us say we choose task 3. This leaves tasks 4 and 5, and we choose 5 first because its depth is higher. This gives a complete schedule of $\langle 0, 1, 2, 3, 5, 4\rangle$, whose tmb_cost is three for task 0, 4 for task 1 and 2 for task 2, which is 9.

Thus, at the end of the first iteration, the Heuristic algorithm chooses task 1 and the second iteration begins. Note that the complete schedules calculated in the first iteration are now discarded and new complete schedules will be built in the second iteration, all of which will have task 1 scheduled first.

Fig. 3.15 summarizes the way in which the heuristic algorithm traverses the candidate search graph using Fig. 3.11 as input. As we described above, in the first iteration, two complete schedules are built, one starting with task zero and one starting with task 1. The latter is chosen by the heuristic algorithm, indicated by the bold arrow. The tmb_cost is also shown in the figure; note the cost of 9 if we choose task 0, versus the cost of 8 if we choose task 1. In the second iteration, the heuristic algorithm considers tasks 0, 2 and 4, and computes the corresponding three depth-first schedules. Choosing task 4 next is the best option. After task 4 has been selected, the algorithm computes two new depth-first complete schedules corresponding to adding tasks 0 and 2, respectively, to the existing partial schedule of $\langle 1, 4\rangle$. Adding task 2 is cheaper, as shown in the figure. The complete schedule generated by the heuristic algorithm is indicated by the bold arrows: $\langle 1, 4, 2, 0, 3, 5\rangle$. Its tmb_cost is 6.

The intuition behind computing complete schedules in a depth-first manner is to schedule successors right after their ancestors; notice that when a task with a higher depth than the previous task is chosen, these two tasks should be very close together in the topological sort of the precedence graph. However, any other heuristic for building possible complete schedules for a given schedule prefix is compatible with the framework we have described in this section.

Finally, we discuss the time complexity of the heuristic algorithm. Pre-processing the precedence graph to compute depth information (i.e. longest paths to a root in G) can be done via a linear-time shortest-paths algorithm on G with negative edge weights (this is only possible because G is a DAG, where after edge weight negation no negative cycles are possible). Now, one iteration of the algorithm involves computing multiple complete schedules in a deepest-first manner. For each such complete schedule, exactly one task is moved

from the schedulable set to the actual schedule, and the successors of this task are added to the schedulable set. Therefore, each node and edge in G need to be visited only once. If the set of schedulable tasks is maintained in a data structure such as a binary heap that allows retrieval and deletion of the minimum-depth node and insertion in $\mathcal{O}(\log|V|)$, computing one complete schedule requires $\mathcal{O}(|E|+|V|\log|V|)$.

The overall heuristic algorithm iterates $|V|$ times. In each iteration, there are at most $|V|$ schedulable tasks, each of which requires a complete schedule to be built, at a cost of $\mathcal{O}(|E|+|V|\log|V|)$, and its tmb_cost must be computed, but the former dominates the runtime. This gives the overall runtime complexity of the heuristic algorithm as $\mathcal{O}(|V|\cdot|V|\cdot(|E|+|V|\log|V|))=\mathcal{O}(|E|\cdot|V|^2+|V|^3\log|V|)$.

The code to run all the above algorithms can be found on github at `https://github.com/arbaer/schedule`.

3.3 `RepoSim`: a simulator to assist the fine-tuning of repository performance

`repoSim` is an ns2-based simulator aimed at assisting the fine-tuning of mPlane repository performance. From a teletraffic point of view, the mPlane repository can be seen as a data center network where jobs of different type compete for the same physical resources (CPU, storage and bandwidth). Depending on how the repository network is managed, this architectural implication has possible consequences not only on the timeliness of the results (e.g., results stuck behind a large transfer), but also possibly about the accuracy of the results themselves (e.g., control messages in iterative drill-down analysis slowed down by bulk transfer) and need careful investigation.

The overall goal of repoSim is to use simulation as a preliminary, necessary step to investigate a broad spectrum of solutions for traffic management in mPlane repositories. As output of this phase, hopefully a few candidate solutions will emerge that are worth implementing in real operational mPlane repositories. The need for a tool such as repoSim can be clarified considering a few illustrative examples, that are worth making to better cast the above general observations and flow taxonomy to the mPlane use cases.

3.3.1 repoSim mPlane model

Being a simulator, repoSim does not pretend to fully model all mPlane system details. Rather reposSim has a precise simplified view of mPlane as a holistic system generating a workload that insists (or, stresses) a peculiar data center network: namely, the mPlane repository network.

With this goal in mind, repoSim embeds a simplified view of the mPlane control and data flows crossing a repository. This view is depicted in Fig. 3.16, that represent a general mPlane workflow, valid for both active or passive measurements. The picture shows a reasoner, or intelligent user, interacting with mPlane through a supervisor, triggering WP2 active/passive measurement nodes [yellow arrows], that generates a workflow that will solicite WP3 repositories.

While repoSim does not aim at fully implementing intelligent reasoners, users, probes, etc., it is however aware that each of the above mPlane components pose a different stress to the architecture. As such, as opposite to a microscopic detailed view of the mPlane system, repoSim takes a more narrow view of the mPlane repository component. The repository is then represented as a data center network, whose interaction with other mPlane components is abstracted in terms of a heterogeneous mix of network flows, each having specific requirements (e.g., throughput vs delay), as we detail next.

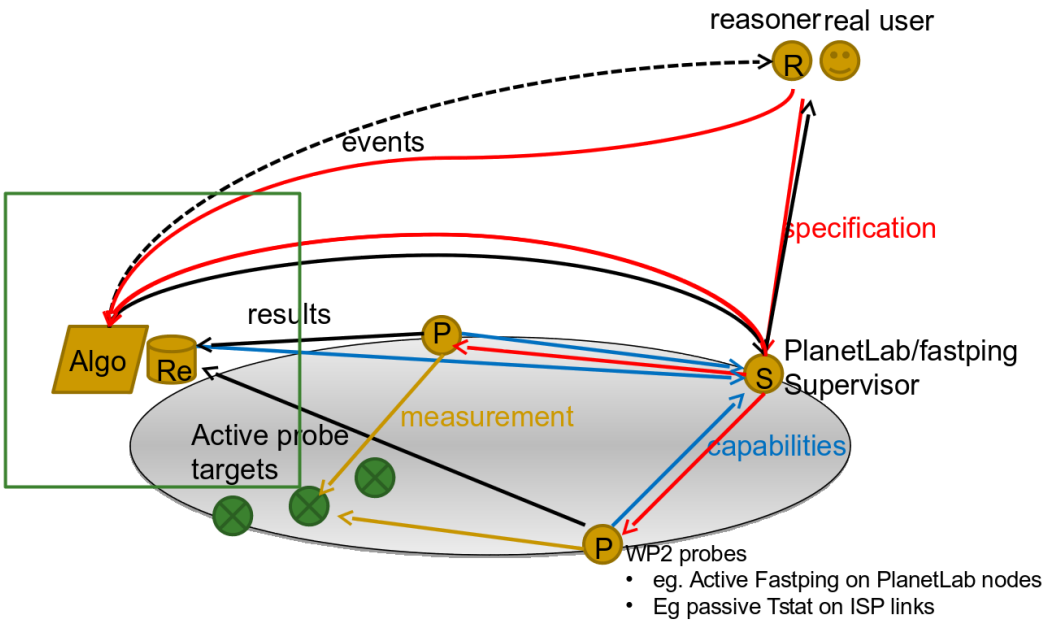


Figure 3.16: repoSim: holistic view of mPlane architecture, with a focus on the repository (boxed)

3.3.2 repoSim flow taxonomy model

As depicted in Fig. 3.16, a mixture of flows insists on mPlane repository: i.e., flows that *enter or exit* the repository, or even flows that are *confined within* the repository data center. Specifically, interactions of mPlane components (e.g., intelligent reasoners, users, supervisors, probes, etc.) are abstracted by means of several kind of flows:

- store raw data (eg CSV, binary, ...) [thick black arrow]
- access raw data (eg FTP, HTTP, ...) [thick black arrow]
- export raw data (eg IPFIX, ...) [thick black arrow]
- receive/issue instructions (e.g., specification) [thin red arrow]
- cooking data to some extent according to received instructions (e.g., MapReduce, or other algorithms) [thick gold arrow]
- generate results and events (i.e., outcome of the above) [dotted black arrow]
- state all the above (i.e., capability) [thin blue arrow]

For the sake of clarity, with reference to the large-scale data analysis algorithms introduced in the earlier section, these data flow may include features pertaining to user of Web browsing sessions, or video streaming sessions, or Cloud services, possibly over mobile phones. In case of passive analysis, feature vectors may have the form of per-flow logs (e.g., Tstat logs) that need to be periodically moved to the repository networks (e.g., through Tstat log_sync), generating large volumes of data [incoming thick black arrows].

Depending on the use case, the instantiated Intelligent reasoner workflows, can further trigger additional active measurement, as for instance when iterative drilling for root cause analysis in the Cloud troubleshooting use case [incoming thick black arrows].

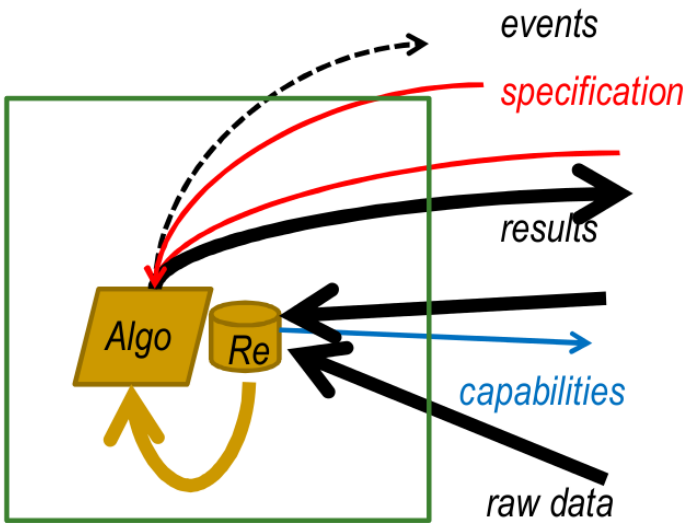


Figure 3.17: repoSim: simplified view of mPlane control and data flows crossing a repository

In case a repository exports multiple analysis capabilities [thin blue arrow], it also possibly receives triggers (i.e., mPlane specifications) to run different analysis in parallel [thin red arrow]. Results of the analysis can either be exported in raw format [outgoing thick black arrow] or more likely in forms of events [outgoing dotted black arrow].

3.3.3 repoSim motivations

Narrowing down our focus to the repository and simplifying even further, from a system point of view, it emerges that WP3 large-scale data analysis involves a handful of concurrent flow types. Such flows are either confined within the repository itself, or cross its interface toward other parts of mPlane infrastructure (or external networks).

These flows share the same physical medium, i.e., the repository network: as multiple reasoners can run their analysis concurrently on the same repository, statistical multiplexing not only involves flows of a single use case. Additionally, even for a single use case, control and data workflows are intermingled, and possibly interdependent (e.g., since a data transfer/processing is not launched until the corresponding specification is not received and parsed by the repository).

It should also be clear that these flows are rather heterogeneous in their size (big data vs short control flows) and consequently requirements (high throughput vs low delay respectively). Indeed, fat data transfers either within (map phase of a MapReduce job, gold arrow) or across (data import or export) the repository network are intermingled with short control flows (incoming specifications, outgoing events and capabilities).

Whereas data transfers need to be optimized for throughput, short control exchanges privileges low delay. Since both kinds of flows coexist on the mPlane repository, it is imperative to efficiently manage the traffic in the repository network. We explain the reason why traffic management in the repository network is crucial for mPlane with the help of Fig. 3.18, that illustrates the requirements and interdependence of short vs long flows.

In more details, Fig. 3.18 shows a dual interdependence of the short vs long transfers. Short transactions are used in mPlane repositories to either advertise capabilities of data processing, instantiate new data processing, and timely exporting crucial results of these processing via events. Long transactions are instead either useful to import/export data to/from mPlane repositories, or to move data within the repository network during data processing. The typical workflow is thus that a short transfer (specification) will trigger a

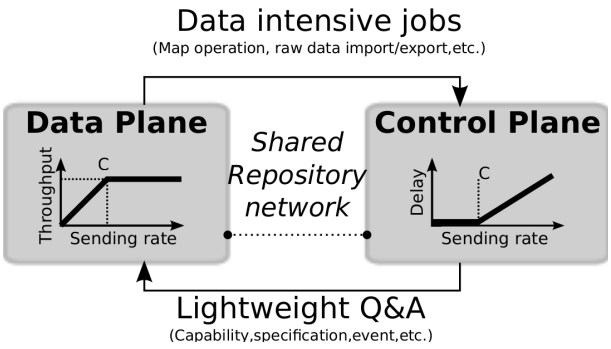


Figure 3.18: repoSim: synopsis of short vs long flows requirements and interdependence.

long one (data import/export/processing, etc.), whose completion will possibly trigger short transfers again (events, etc.).

Intra use-case dependency is tied to the fact that these transactions happen in sequence, so that for instance a slow data transfer rate may delay the reception time of events. Inter use-case dependencies instead arise when, for instance, a specification of a new processing request, or an event message at the end of a data processing phase, queues behind a large transfer, delaying the start or end times respectively. In case of iterative workflows, these delays may accumulate and harm the system responsiveness.

From a generic teletraffic point of view, we argue that systems should be optimized for the data plane throughput, as long as this does not hurt the control plane delay. However, as shown in Fig. 3.18, these two objectives tradeoff: when the sending rate exceeds the system capacity C, the throughput saturates, but the delay grows due to buffering. When the sending rates falls below C, short transactions complete timely, but the data transfer is longer due to reduced efficiency. Hopefully, repoSim can assist tuning this delicate tradeoff.

3.3.4 repoSim modules

Abstracting from the mPlane workflows, from a teletraffic point of view, we can model this resource sharing as a multiplexing of different flows of type, size, and load, both within and enter/exit the repository infrastructure, as examplified in Fig. 3.19 where arrows have different sizes.

Under this light, the challenge is to design, implement and evaluate an efficient repository network to support mPlane operations. As previously explained, the goals of the design are mainly:

- Sustained throughput to avoid slowdown of data cooking (e.g. elephant MapReduce data transfer in a map phase)
- Low-delay communication for short transactions (e.g. mice control flows)

To achieve these goals, several design aspects can be considered, which include planning decisions, hardware implementations and software operating at multiple layers:

- topology design (e.g., FatTree vs BCube vs Jellyfish, etc.)
- L2 scheduling (e.g., Stocastic Fairness Queueing, fq_codel, etc.)
- L2/3 switching and routing (e.g., Spanning tree, TRILL, ECMP, etc.)
- L4 congestion control (e.g., TCP, DCTCP, MPTCP, low-latency TCP, transaction TCP to avoid 3-way handshake for short flows, etc.)
- L7 application layer solutions (e.g., MapReduce schedulers, RepFlow to replicate short flows, etc.).

It appears that the overall issue of mPlane repository management is larger than the scope of what can be done within a single class. That is to say, while it is imperative to schedule jobs within the MapReduce framework, this L7 scheduling knob will only enforce fairness among MapReduce jobs, but will otherwise left untouched the problem of short transfer congestion behind MapReduce jobs: for the perspective of a short transaction, being queued behind one or another MapReduce jobs will be perfectly equivalent.

Notice also that only recently the datacenter community has started investigating the use of joint design choices at multiple levels, so their analysis is a green field so far. repoSim aims at assisting the design, by e.g., allowing to understand which of the above ``ingredients'' would make a successful ``recipe'' for a mPlane repository network.

For the sake of the example, let us consider scheduling at L2 vs scheduling at L7: repoSim could assist in evaluating the benefits of each approach alone, or their combination. This is particularly interesting, as during mPlane we already have discovered unexpected and negative interactions of uncoordinated design choices having the same goal (notably, a vicious interaction between low-priority L4 congestion control and L2 scheduling). repoSim would allow avoiding selecting solution for a repository network that could exhibit such undesirable behavior. Ultimately, it hopefully assists in selecting an mPlane repository architecture that tradeoffs implementation complexity and performance.

3.3.5 Results

repoSim simulates a combination of flows transferring simultaneously in a mPlane repository data-center network. It allows user to specify parameters of different components involving network performance, such as the network topology configuration, flow size and arrival pattern, transport protocol, and scheduling protocol. As output, we measure the flow completion time, that is directly relevant for short flows, as well as for long transfer (since low completion time implies high throughput).

In the current stage, we are simulating generic workloads from the literature, to have a broad comparison of our new solutions that is easier to cross compare with the state of the art. An example of the performance gathered with repoSim is shown in Fig. 3.19 that shows state of the art results (taken from pFabric[3], in the left) and our own comparison of the state of the art with alternative architectures. Specifically, the picture on the right contrast the pFabric[3] state of the art (that as can be seen in the left hand side pictures, obtains near optimal performance) with very simple scheduling solutions that only act at L2 (namely a SFQ scheduling discipline vs a RED active queue management solution). As a function of a growing data center load on the x-axis, the picture shows the average flow completion time, normalized to the ideal one (so that a FCT of 1 implies optimal performance).

As it can be seen from our preliminary results, *simple L2 solutions, such as SFQ, that have been so far neglected by the data center community, are within a factor of 2 from the optimal performance*, which allows to expect that a reasonably simple design for mPlane repository that also have near optimal performance can be found. On the long run, the aim would be to perform simulations with real mPlane workload (e.g.,trace driven taken from real mPlane repository usage). This will be instrumental to fine tune and benchmark the real mPlane repository under real workloads.

A preliminary version of this tool, with a guide on how-to use it, is available from the project website at `https://www.ict-mplane.eu/public/reposim`.

3.4 Performance of computing platforms

To process the data coming from the probes, the requests from the reasoner and the algorithms as specified in the analysis modules of each use case, the repository has to implement enough computational power. To achieve that, several computing platforms are available: examples are Hadoop, which is oriented to the off-line processing of data, and Storm, which is oriented to the on-line (stream) processing of data.

To investigate *what are the capabilities of the current data-analytics platforms in processing unbound stream of*

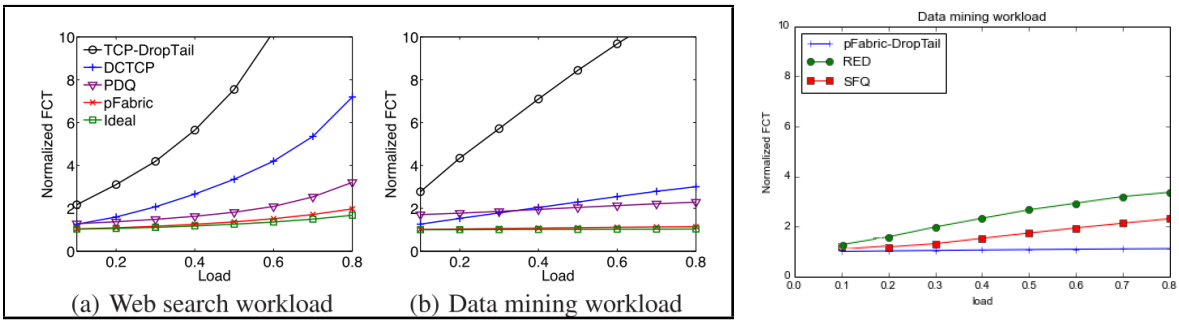


Figure 3.19: repoSim: performance evaluation contrasting existing data center architectures (left, state of the art from [3]) with classic scheduling schemes in repoSim (right).

data, we have analyzed the performance we achieve when running applications on the two stream-processing platforms Apache S4 and Storm, and on BlockMon [21], a platform originally designed for running packet processing operations on a single multi-core node, which we extended to execute applications which are distributed across machines.

To this end, we implemented on all platforms two stream-monitoring applications: *VoIPSTREAM* [12], a phone anomaly detection system, and *Twitter trending*, a system that monitors topics discussed by Twitter users over time. In the former case, we investigate how the platforms help speed up the execution of applications that need high-computational power: the application is designed to one probe sending (dispatching) data to multiple aggregation point. In the latter case, we focus on how the platforms handle data transfer over the topology: the application is designed to have multiple probes sending data to one aggregation point. Please refer to [42] for a complete description of the applications and of the platforms. As a matter of fact, multiple probes and multiple aggregation points might concur to form any service-monitoring application: by assessing how the distributed streaming platforms perform in each separated scenario, we aim at gaining insight on how such platforms perform in a broad range of applications, such as the ones considered under the mPlane umbrella.

We release distributed BlockMon to the public [13]. A code release is also available from the mPlane webpage at `https://www.ict-mplane.eu/public/blockmon`: in this release, we make available some applications being used for testing the performance of BlockMon against the other stream-computing platforms, together with the necessary blocks for executing distributed applications, that is, importer and exporter blocks for exchanging data across machines.

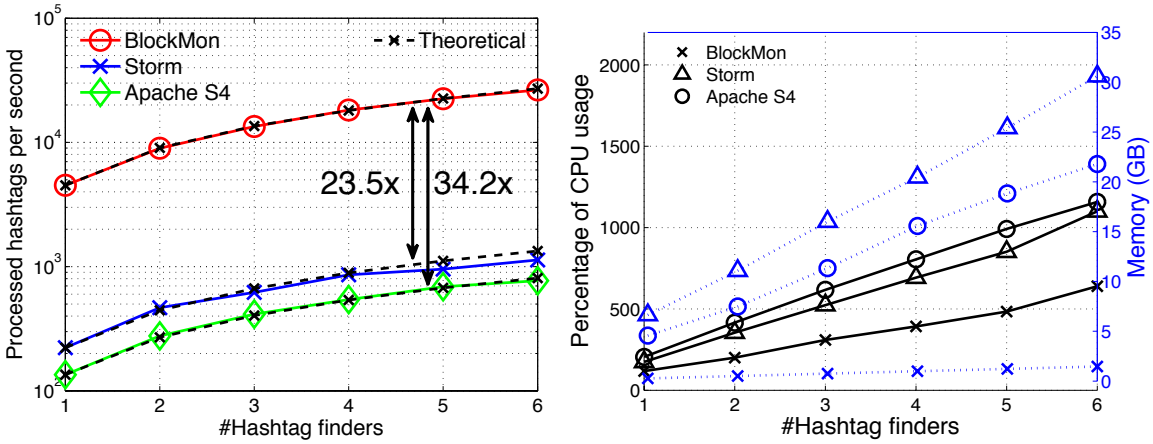
3.4.1 Experimental analysis

Given the target of assessing performance, we assume no failure during the experiments and we focus on scalability and costs in terms of CPU and memory usage.

Our testbed is composed of 14 commodity machines, each one hosting two AMD Opteron(tm) Processors 246 (single core) and 4GB RAM. A 16-port switch connects the 1GbE interfaces of all machines. We estimate the value of each machine to be around $1000 on today's market.

For the *Twitter trending* application, we considered a dataset composed of around 20 millions of tweets, in the JSON format as provided by GNIP. As for *VoIPSTREAM*, we used a dataset composed of few tens of million of anonymized Call Detail Records (CDR) collected over a period of several consecutive weeks, thanks to the collaboration of a small European telecom operator.

Figure 3.20(a) shows the performance of *Twitter trending* as the number of the hashtag finders (HF) increases. On all the platforms, the application scales linearly, with a gain in performance of 23.5x (34.2x) when we use BlockMon compared to Storm (Apache S4). Note that this scaling behavior is expected: as by



(a) *Twitter trending*: scalability (y-axis is in log scale). (b) *Twitter trending*: total CPU (solid line) and memory (dotted line) usage.

design we increase the number of both tweet sources and HFs in the topology, the rate of processed hashtags must increase linearly as long as network capacity towards the hashtag counter is not a bottleneck. In the figure, we report the theoretical behavior for each platform as a dashed black line. Figure 3.20(b) shows the total cost of memory and CPU required by the three platforms to run *Twitter trending*. As all machines in our testbed are identical, we computed the overall memory and CPU usage as the sum of the resources used on each machine. The CPU load of the hashtag counter is even in the worst case (with six HFs on Storm) always below 4%, thus suggesting that one is enough to cope with multiple HFs, which account in turn for around 75% of the total CPU resources.

When testing the computing platforms with the *VoIPSTREAM* application (not shown here, details in [42]), we observe that the processing rate with BlockMon is up to 2.5x faster than Storm. As for Apache S4, we observed that the bottleneck is due to the communication between the adapter and the cluster where the application runs, which prevents the application from scaling at all: in this case, *VoIPSTREAM* on BlockMon runs up to 11.2x faster. Interestingly, we were not able to run Apache S4 on the whole testbed: under high-memory consumption cases, the communication between the node and ZooKeeper hungs, thus partitioning the cluster. Developers of Apache S4 are aware of this issue.

Our results point out that existing stream-processing platforms have serious issues when it comes to performance, which are not due to mechanisms for high availability or dynamic message routing: improving performance is possible, and our enhanced BlockMon showed that. We believe that our findings can help improve existing architectures to target stream data processing for network *stream monitoring*.

4 Conclusions

In this deliverable we have described the progress on the design and implementation of scalable algorithms that operate on very large amounts of data. To make things more practical, those algorithms have been presented in the context of each use case, and their results have been shown.

A core part of this deliverable has focused on the design, implementation and evaluation of scheduling protocols for the efficient and fair allocation of computing resources to network data analysis jobs. Essentially, we have proposed new components for scheduling analytic tasks in parallel processing frameworks by considering the particular computational workloads generated by the mPlane infrastructure. As a result of this, within mPlane we have realized three tools that can be used at the repository: *Hadoop Fair Sojourn Protocol*, a scheduler for Apache Hadoop; *schedule*, a tool for cache-oblivious scheduling of shared workloads; and *repoSim*, a ns2 based simulator to fine-tune the mPlane repository performance. For each of them we have provided documentation and a software release through the official mPlane website. For more information about it, please visit `http://www.ict-mplane.eu/public/software`.

Furthermore, we have provided a performance comparison of three distributed stream-computing platforms, by comparing our open-source platform Blockmon against the platforms Storm and Apache S4: in both the applications being tested, Blockmon showed to perform better.

As future directions, we plan to continue our work in the implementation of large scale data analysis algorithms, and the integration of the tools that have been designed in this phase into the mPlane architecture.

References

[1] CDH 4 installation guide -- tips and guidelines.

[2] Page replacement design.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM*, pages 435--446, 2013.

[4] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: Coordinated memory caching for parallel jobs. In *USENIX NSDI*, 2012.

[5] Apache. Hadoop. `http://hadoop.apache.org/`.

[6] Apache. Hadoop fair scheduler. `http://hadoop.apache.org/docs/stable/fair_scheduler.html`.

[7] Apache. Hadoop MapReduce JIRA 1184. `https://issues.apache.org/jira/browse/MAPREDUCE-1184`.

[8] Apache. PigMix. `https://cwiki.apache.org/PIG/pigmix.html`.

[9] Apache. Spark. `http://spark.incubator.apache.org/`.

[10] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2013.

[11] A. Bär, P. Casas, L. Golab, and A. Finamore. DBStream: an Online Aggregation, Filtering and Processing System for Network Traffic Monitoring. In *Proceedings of 5th International Workshop on TRaffic Analysis and Characterization, Nicosia, Cyprus*, page 6. IEEE Computer Society, Aug. 2014.

[12] G. Bianchi, N. d'Heureuse, and S. Niccolini. On-demand time-decaying bloom filters for telemarketer detection. *Comput. Commun. Rev.*, 41(5):5--12, Sep. 2011.

[13] BlockMon. http://blockmon.github.com/blockmon (accessed 2012-11-10).

[14] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly & Associates Inc, 2005.

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

[16] Y. Chen, A. Ganapathi, R.Griffith, and R. Katz. The case for evaluating MapReduce performance using workload suites. In *Proc. of IEEE MASCOTS*, pages 390--399, 2011.

[17] E. G. Coffman and P. J. Denning. *Operating systems theory*, volume 973. Prentice-Hall, 1973.

[18] P. Crescenzi and V. Kann. A compendium of np optimization problems, 1998. `ftp://ftp.nada.kth.se/Theory/Viggo-Kann/compendium.pdf`.

[19] M. Dell'Amico, D. Carra, M. Pastorelli, and P. Michiardi. Revisiting size-based scheduling with estimated job sizes. *CoRR*, abs/1403.5996, 2014.

[20] P. J. Denning. Thrashing: Its causes and prevention. In *Fall Joint Computer Conference*. ACM, 1968.

[21] A. di Pietro, F. Huici, N. Bonelli, B. Trammell, P. Kastovsky, T. Groleat, S. Vaton, and M. Dusi. Blockmon: Toward high-speed composable network traffic measurement. In *Proceedings of the IEEE Infocom Conference (mini-conference)*, 2013.

[22] E. J. Friedman and S. G. Henderson. Fairness and efficiency in web server protocols. In *SIGMETRICS Performance Evaluation Review*, volume 31, pages 229--237. ACM, 2003.

[23] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[24] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. In *Theoretical Computer Science 1(3)*, pages 237--267, 1976.

[25] L. Golab, J. S. S. T. Johnson, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD*, pages 847--854, 2009.

[26] S. Gorinsky and C. Jechlitschek. Fair efficiency, or low average delay without starvation. In *ICCCN*, pages 424--429. IEEE, 2007.

[27] M. Harchol-Balter. Queueing disciplines. *Wiley Encyclopedia of Operations Research and Management Science*, 2009.

[28] M. Harchol-Balter et al. Size-based scheduling to improve web performance. *ACM TOCS*, 21(2):207--233, 2003.

[29] K. Hayashi and R. Fujimaki. Factorized asymptotic bayesian inference for latent feature models. In C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, editors, *NIPS*, pages 1214--1222, 2013.

[30] A. A. I. Psaroudakis, M. Athanassoulis. Sharing data and work across concurrent analytical queries. In *PVLDB 6(9)*, pages 637--648, 2013.

[31] L. Kleinrock. *Theory, volume 1, Queueing systems*. Wiley-interscience, 1975.

[32] S. A. Klugman, H. H. Panjer, and G. E. Willmot. *Loss models: from data to decisions*, volume 715. John Wiley & Sons, 2012.

[33] D. Lu, H. Sheng, and P. Dinda. Size-based scheduling policies with inaccurate scheduling information. In *MASCOTS*, pages 31--38. IEEE, 2004.

[34] J. Nagle. On packet switches with infinite storage. *IEEE TCOM*, 35(4):435--438, 1987.

[35] C. Papadimitriou. The np-completeness of the bandwidth minimization problem, computing, 16(3). pages 263--270, 1976.

[36] M. Pastorelli, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi. HFSP: size-based scheduling for Hadoop. In *Big Data*. IEEE, 2013.

[37] M. Pastorelli, M. Dell'Amico, and P. Michiardi. Os-assisted task preemption for Hadoop. In *Proc. of DCPerf*, 2014.

[38] M. Pastorelli et al. Practical size-based scheduling for MapReduce workloads. *CoRR*, abs/1302.2749, 2013.

[39] I. A. Rai, G. Urvoy-Keller, and E. W. Biersack. Analysis of LAS scheduling for job size distributions with high variance. *SIGMETRICS Performance Evaluation Review*, 31(1):218--228, 2003.

[40] L. E. Schrage and L. W. Miller. The queue M/G/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670--684, 1966.

[41] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM TOIT*, 6(1):20--52, 2006.

[42] D. Simoncelli, M. Dusi, F. Gringoli, and S. Niccolini. Stream-monitoring with blockmon: convergence of network measurements and data analytics platforms. *SIGCOMM Comput. Commun. Rev.*, 43:29–36, 2013.

[43] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM TON*, 6(5):611--624, 1998.

[44] TPC. Tpc benchmarks. `http://www.tpc.org/information/benchmarks.asp`.

[45] D. Tsaih, G. Wu, C. Chang, S. Hung, C. Wu, , and H. Lin. An efficient a* algorithm for the directed linear arrangement problem. In *WSEAS Transactions on Computers, 7(12)*, pages 1958--1967, 2008.

[46] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop Yarn: Yet another resource negotiator. In *SoCC*. ACM, 2013.

[47] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[48] A. Wierman. Fairness and scheduling in single server queues. *Surveys in Operations Research and Management Science*, 16(1):39--48, 2011.

[49] A. Wierman and M. Nuyens. Scheduling despite inexact job-size information. In *SIGMETRICS Performance Evaluation Review*, volume 36, pages 25--36. ACM, 2008.

[50] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. of ACM EuroSys*, pages 265--278, 2010.