Master thesis in Global Software Engineering

Design and implementation of a
MLFQ scheduler for the Bacula backup software

Paolo Di Francesco

Email: paolodifrancesco85@gmail.com

IDT supervisor  Ivica Crnkovic  Email: ivica.crnkovic@mdh.se
UDA supervisor  Vittorio Cortellessa  Email: vittorio.cortellessa@univaq.it
LNGS supervisor  Stefano Stalio  Email: stefano.stalio@lngs.infn.it
IDT examiner  Ivica Crnkovic

Academic year 2011/2012
...Dedicated to those who remain even when they are gone away...
Abstract

Nowadays many organizations need to protect important digital data from unexpected events, such as user mistakes, software anomalies, hardware failures and so on.

Data loss can have a significant impact on a company business but can be limited by a solid backup plan.

A backup is a safe copy of data taken at a specific point in time. Periodic backups allow to maintain up-to-date data sets that can be used for efficient recovery.

Backup software products are essential for a sustainable backup plan in enterprise environments and usually provide mechanisms for the automatic scheduling of jobs.

In this thesis we focus on Bacula, a popular open source product that manages backup, recovery, and verification of digital data across a network of heterogeneous computers. Bacula has an internal scheduler that manages backup jobs over time. The Bacula scheduler is simple and efficient, but in some cases limited.

A new scheduling algorithm for the backup software domain is presented together with an implementation developed for Bacula. Several benefits come from the application of this algorithm and two common issues such as starvation and the convoy effect are handled properly by the new scheduler.

List of Terms: Bacula, backup software, data backup, recovery, scheduling algorithm, MLFQ scheduling, dynamic priority, aging, starvation, convoy effect
# Table of Contents

1. Introduction .......................................................................................................................... 1
   1.1 The Gran Sasso National Laboratory ............................................................................. 2
   1.2 Research problem, contribution and methodology ....................................................... 3
   1.3 Roadmap .......................................................................................................................... 5

2. Backup software ................................................................................................................... 6
   2.1 Data backup ..................................................................................................................... 6
   2.2 Backup software systems ............................................................................................... 8

3. The Bacula backup software ............................................................................................... 9
   3.1 Overview .......................................................................................................................... 9
   3.2 Bacula design .................................................................................................................. 10
   3.3 Client/server architecture .............................................................................................. 11
   3.4 Job configuration ............................................................................................................ 12

4. Background on scheduling algorithms .............................................................................. 13
   4.1 First-come, first-served scheduling ............................................................................. 14
       4.1.1 The convoy effect .................................................................................................... 14
   4.2 Priority scheduling .......................................................................................................... 15
       4.2.1 Starvation and aging .............................................................................................. 16
   4.3 Multilevel queue scheduling ......................................................................................... 17
   4.4 Multilevel feedback queue scheduling ......................................................................... 18
7.3 Scheduler comparison...........................................................................................................50
7.4 Tuning guidelines..................................................................................................................55
7.5 Preliminary analysis in a real environment............................................................................57

8. Conclusions..................................................................................................................................58
  8.1 Future works..........................................................................................................................60
  8.2 Code evolution.........................................................................................................................61

References.......................................................................................................................................62

Acronyms.......................................................................................................................................65

APPENDIX A: Original scheduler pseudo-code..........................................................................66

APPENDIX B: New scheduler pseudo-code..................................................................................67

APPENDIX C: Backup software products web pages .................................................................77
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bacula main components</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>Static and dynamic priority</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>New job behaviors</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Example 1: High priority jobs planned start</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>Example 1: Jobs delayed by a lower priority job</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>Periodic job</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Example 2: Job with different aging values</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>Example 3: Original scheduler plan</td>
<td>52</td>
</tr>
<tr>
<td>9</td>
<td>Example 3: New scheduler plan</td>
<td>52</td>
</tr>
<tr>
<td>10</td>
<td>Example 3: Waiting time comparison</td>
<td>54</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Example 3: Jobs planned start, duration and priority ..................................................51
Table 2: Example 3: Jobs start time ..........................................................................................51
Table 3: Example 3: Job waiting times ......................................................................................54
Table 4: Acronyms ....................................................................................................................65
Table 5: Free backup software products ....................................................................................77
Table 6: Proprietary backup software products ........................................................................77

Pseudo-code Listings

Listing 1: Original scheduler pseudo-code: Main procedure ..................................................66
Listing 2: New scheduler pseudo-code: Main procedure ..........................................................68
Listing 3: New scheduler pseudo-code: Update lists procedure ..............................................71
Listing 4: New scheduler pseudo-code: Aging procedure .........................................................72
Listing 5: New scheduler pseudo-code: Find job procedure ....................................................74
Listing 6: New scheduler pseudo-code: Schedule recovery procedure ....................................76
Acknowledgements

There are many people I want to thank for the help received in the realization of this work.

First, I would like to thank Stefano Stalio, my supervisor at LNGS laboratories, who has contributed with considerable time and effort to the production of this work.

Second, I want to thank professors Ivica Crnkovic, from the Mälardalen University, and professor Vittorio Cortellessa, from L'Aquila University, who have supervised my work and have helped me to improve its overall quality.

I want to thank professor Henry Muccini, from L'Aquila University, who has given me the opportunity to participate to the Global Software Engineering European Master (GSEEM) program.

Words can not express how grateful I am to my family. Their constant and endless support helped me to reach many important goals, this being only the most recent.

Last but not least, I want to thank all my friends who always deserve a special note in every single step I take.

Thank you all!!!

September 2012
1. Introduction

Backup software products become every day more and more important. In environment where backups are regularly performed their use improves the overall data maintenance process and also the system resource utilization. Data compression, data encryption, data verification, data deduplication, remote backup, media spanning are only a few of the powerful features offered by these products.

While in small environments backups are manually triggered by the same system users, automatic scheduling of backup jobs is necessary when dealing with periodic backups of several hosts.

The majority of the backup products offer priority-based mechanisms for the automatic scheduling of jobs over time, but they rarely deal with two common issues known as job starvation and convoy effect. More sophisticated scheduling capabilities become particularly important when many clients need to be backed up and the resources are limited.

In this thesis, a different scheduling algorithm for the backup software domain is proposed, and the benefits coming from its application are discussed. If properly used, a scheduler based on the new algorithm can obtain better resource utilization and a better system workload balance.

An implementation of this algorithm has been developed for Bacula, an open source backup software product, whose built-in scheduler already manages the scheduling of backup jobs over time. The new scheduler is meant to provide an alternative to, and possibly replace, the original Bacula scheduler, which is simple and efficient but in some cases limited.

The Gran Sasso National Laboratory (LNGS) where the Bacula software has been adopted for data backup since year 2009 is the scenario. During these years, the daily use of this software has triggered the interest of the LNGS Information Technology department towards the development of some improvements to the Bacula scheduler. Most of them were related to
desirable job behaviors that could not be obtained using the Bacula internal scheduler and more in general were not fully supported by any of the backup software products available in the market.

This work only focuses on the scheduling of jobs in a backup plan. All the other aspects of data backup management are beyond the scope of this discussion.

### 1.1 The Gran Sasso National Laboratory

The Gran Sasso National Laboratory (LNGS)\(^1\) is one of four Italian Institutes for Nuclear Physics (INFN) national laboratories.

“It is the largest underground laboratory in the world for experiments in particle physics, particle astrophysics and nuclear astrophysics and its mission is to host experiments that require a low background environment in the field of astroparticle physics and nuclear astrophysics and other disciplines that can profit of its characteristics and of its infrastructures. Main research topics of the present programme are: neutrino physics with neutrinos naturally produced in the Sun and in Supernova explosions and neutrino oscillations with a beam from CERN (CNGS program), search for neutrino mass in neutrinoless double beta decay, dark matter search, nuclear reactions of astrophysical interest.” [12].

A LAN network connects most of the LNGS computers. The Bacula backup software is used to manage backup and restore operations. The two main reasons why an open source software was chosen are: the *open data format*, which allows for long term data availability and tool abstraction, and the fact that no effort has to be put in keeping constantly up-to-date *software licenses*.

The LNGS IT infrastructure is made of heterogeneous computers, ranging from personal laptops, to office computers, to big computation servers.

The broad variety of backup data sets makes the LNGS laboratories a suitable environment for the development and the validation of the new scheduler.

---

1.2 Research problem, contribution and methodology

The importance of runtime job management is often an underestimated feature in the backup domain. Performing backup jobs often require considerable amount of time and storage resources, and in systems where many hosts are regularly backed up an efficient scheduling strategy is necessary: if resources are limited and many jobs are waiting for resource allocation it is of great importance to select at runtime which job is the most important to run. The selection of the most appropriate job to run in a specific moment must not only take into account the importance of the file set to back up, but should also consider how much time a job has been waiting for the resource allocation.

Several difficulties arise for system administrators when they need to set up a backup strategy. A backup plan not only deals with the definition of the frequency and the level of backup jobs, but also with situations where hosts are unavailable, storage resources are inappropriately balanced, and delays are introduced by long jobs.

Job prioritization is the only mean that administrators have for differentiating jobs into categories.

Most backup software products do not offer scheduling capabilities that allow for sophisticated job management strategies. First-come-first-served or fixed-priority strategies that are adopted in the backup domain may result inadequate in conditions of heavy system workload.

This thesis has been developed using a construction methodology. At first, existing software backup products have been analyzed to identify the limitations related to backup job scheduling, and then a new scheduling strategy has been elaborated in order to overcome these limitations. Finally, a new experimental scheduler has been developed within the Bacula software.

The new scheduler has been tested and is currently in use in one of the production chains of the LNGS laboratories. The features introduced with the new scheduler have significantly changed the previously defined backup plan, thus offering important capabilities for a better balance of the system resources.
The Bacula software development team has been informed about the new scheduler and agreed on the possibility to include it in one of the upcoming releases.

The time spent for the analysis of the state of the art in the software backup domain, for the planning of the new job characteristics, for the definition of the scheduling algorithm and for the development of the new scheduler was approximately 7 months, full time. One more month was necessary for the initial testing of the new scheduler and the debugging of the source code. Afterwards, we set up the new implementation for one of the two LNGS production chains, where it is in use since July 2012.

The new scheduler was implemented in the C programming language, and just its module required about 3000 lines of source code. Other modules of the Bacula software were also modified to allow the extensions necessary to the new job directives defined in the algorithm. These changes involved mainly Bacula configuration files and parsing procedures.

Preliminary information has been gathered to provide a first analysis about the benefits achievable with the new scheduler, but statistically relevant data can only be available after a longer time period and we will not be able to provide a measure in time for this thesis.

Next to the production chain, we have used a test environment. This environment has been mainly used for running verification and validation tests on the new scheduler. As soon as the testing phase will be ultimated, the test environment will be used to perform a detailed analysis of the new scheduler behavior under conditions of heavy system workload alternate with conditions of low system workload. This will allow us to define:

- A measurement of the benefits achieved by the scheduler by analyzing the job waiting times, and comparing the jobs average waiting time of both high priority and low priority jobs;
- Analyze the global effects of the aging process;
- Make a precise estimation of the scheduler configuration parameters;
- Evaluate the impact of the early start feature in situations of irregular system workload.
1.3 Roadmap

After this introductory chapter, this thesis is organized as follows.

Chapter 2 deals with the concept of backup, the backup planning activity and the importance of using software products to improve the backup plan.

Chapter 3 gives an overview of the Bacula backup software and highlights its main characteristics and features. Its design and its client/server backup architecture are also described.

Chapter 4 describes the general scheduling algorithms relevant to the discussion. In this chapter job starvation, convoy effect and aging are defined.

Chapter 5 is an overview of the scheduling features offered by different backup software products available in the market. The Bacula scheduling strategy is described in detail. Other domain-related peculiarities are also presented and a general discussion concerning the features common to backup software tools is presented at the end of the chapter.

Chapter 6 discusses the newly proposed scheduler. Section 6.1 describes the new job configuration attributes while the new scheduling strategy is described in section 6.2. The schedule recovery feature is presented in section 6.3.

Chapter 7 discusses the benefits achievable with the use of the new scheduling algorithm. The complexity analysis is reported in section 7.2. Section 7.3 compares the original scheduler with the proposed scheduler, while section 7.4 suggests guidelines to consider when using the new scheduler. Section 7.5 provides a preliminary analysis of the new scheduler behavior, after its deployment in one of the LNGS production chain.

Chapter 8 presents the conclusions, and discusses further enhancement to this work.
2. Backup software

Data backup and backup software systems are introduced in this chapter.

2.1 Data backup

“Backups are snapshot copies of data taken at a particular point in time, stored in a globally common format, and tracked over some period of usefulness, with each subsequent copy of the data being maintained independently of the first.”[13].

Backups are essential for data protection [4] within any organization and are crucial for the recovery of missing or corrupted data.

In the information era the unexpected loss of data can cause problems in the business of a company. Moreover there never is the guarantee that all data can be always recovered. By constantly storing up-to-date backups this risk is reduced, and data recovery becomes an easy and relatively safe process.

The backup strategies in use within organizations differ significantly because of different organization's needs. This activity requires significant amounts of time and resources and must always be carefully planned.

Basically there are three different levels of backups: full, differential and incremental.

A full backup represents a complete snapshot of the data that is intended to be protected and provides the baseline for all other backup levels. Full backups are independent from each other and they enable quick data restoration. To limit the problem of redundant copies of unchanged data in subsequent full backups, administrators can use either differential or incremental backups.

A differential backup identifies and stores only the changes occurred since the last full backup, while an incremental backup captures the changes occurred since the last backup of
any type.

When data needs to be restored, the most recent full backup is used as the main reference point. If more recent differential backups are available, the data is recovered using only the most recent full and differential backups.

When using incremental backups, data restoration requires the backup system to analyze the whole set of incremental backups taken after the latest full or differential.

A strategy based on incremental backups usually results in longer recovery times, due to the latency involved in restoring multiple backup images, but reduces data redundancy, which is a typical inconvenient for strategies based on full and differential backups.

A combination of different backup levels often is the best compromise between performance and data redundancy.
2.2 Backup software systems

A large number of backup software tools provide support for data backup, restore and verification. Even though most of them are proprietary, for example NetBackup, Backup Exec, EMC Networker, and Simpana, there are also a few interesting products distributed over the GPL license, like Bacula, and the BSD license, like Amanda.

Backup software products differ in their architectures and in the features they offer. Every time a product is deployed in an enterprise environment, it needs to be configured in order to meet the desired requirements. The correct design and deployment of the backup system constitutes a fundamental step towards good performance [4] and system scalability [13].

Several backup software products are able to provide multi-platform support, to perform backups of computers distributed over networks, and also to automate the scheduling of backup jobs over time.

A data backup requires mainly time, network and storage resources. Effort must be spent to realize an efficient backup plan, especially when dealing with a consistent number of hosts.

Backup jobs often differ in their characteristics. They range from data-intensive backups (huge amount of data, such as the backup of a file server), to backup jobs involving large number of small files (e.g. the backup of mail boxes on a mail server), to small backups (usually daily backups of workstations).

The scheduling mechanisms underlying each backup software must keep the job schedule consistent and also handle all the problems that may arise, such as backup errors, unreachable nodes, and so on.
3. The Bacula backup software

In this chapter we shortly describe the relevant characteristics of Bacula, its design and its features. Most of the information is extracted from the main documentation [18] and is only an overview of the product.

3.1 Overview

Bacula is a free and open source software developed in the C++ programming language and distributed under the GNU General Public License version 2.0 (GPLv2).

The latest release is the 5.2.11, released on September 11, 2012.

“Bacula is a set of computer programs that permits the system administrator to manage backup, recovery, and verification of computer data across a network of computers of different kinds.”[18].

Bacula has a modular design which guarantees the system scalability to hundreds of computers distributed over a large network. It relies on a network client/server architecture and can either run entirely upon a single computer or be distributed over different machines.

The application of Bacula in a large enterprise requires some initial effort for the configuration and the setup, but afterwards the system is not supposed to need continuous human intervention. Once the software is set for running, the administrator's job is limited to monitoring the correct system operations.

All the principal platforms are supported in Bacula: Linux, Mac OS X, Unix, Windows, and BSD. At the time of writing, the software is fully supported and constantly developed, with frequent releases.
3.2 Bacula design

Bacula is composed of five modules:

- The director,
- The file daemon,
- The storage daemon,
- The console,
- The catalog database.

The director is the system coordinator. It is responsible for the scheduling of jobs and for maintaining the catalog up-to-date. Administrators interface directly with the director through the console module. The internal scheduler for job management is included in the director: the list of jobs to run, their priorities, the scheduled start time and all the runtime information concerning jobs is maintained here.

The file daemon, or client, usually runs as a service on each host that needs backups. This module is responsible for performing operations (backup, restore or verification) on the same host where it is deployed, and communicates with both the director and the storage daemon.

The storage daemon is responsible for storing data on physical media. Many typologies of devices are supported, among all: disks, tapes, cd/dvd, and usb drives. When needed, the storage daemon communicates directly with the director and the file daemon.

The console is the means by which administrators and users control Bacula. It communicates directly with the director via network. Administrators use the console to start or cancel jobs, review jobs output, query or modify the catalog.

The catalog is a SQL database holding information about completed jobs, volumes used, files location, and so on. The catalog allows for rapid data restoration and supports different DBMS.

Usually, all components communicate through the network and are deployed separately.
3.3 Client/server architecture

Bacula is based on a client/server architecture.

When time comes for a backup, the director contacts the client and delivers all the information concerning the operation to perform (backup level, set of files, storage daemon host, etc.). The client, in turn, contacts the storage daemon and starts the operation. The client will not contact the director again until the operation is completed unless errors arise, in which case the director is immediately alerted with an error message.

When performing a backup, the set of files to be backed up is sent from the director to the client. The client knows exactly which files must or must not be backed up and therefore there is no need to further contact the director. The exchange of data concerns only the client and the storage daemon, which in turn informs the director. The director will then maintain the catalog up-to-date.

Figure 1 shows the Bacula components interaction\(^2\) when performing backups.

![Bacula main components diagram]

**Figure 1: Bacula main components**

---

3.4 Job configuration

Each Bacula backup job requires a set of attributes to be defined. A basic configuration must contain at least the following directives:

- name of the client,
- type of job and its level,
- set of files to back up (fileset),
- schedule,
- storage,
- media pool.

The name directive is used by the director to identify the client machine.

The type directive selects whether to perform a backup, a restore, a verification or other kind of operations. When dealing with backups, the level directive defines if the backup is full, differential or incremental.

The schedule directive defines when the job has to be automatically scheduled.

The storage directive defines the name of the storage services where the user wants to back up a specific fileset.

The pool directive defines the pool of volumes where data must be stored.
4. Background on scheduling algorithms

Scheduling is the process of deciding how to assign resources to different tasks. A scheduling algorithm, or scheduling policy, is the sequence of steps that the scheduler makes to perform these decisions.

In this treatment the words job, process, and backup are used to indicate any job that a backup software runs.

The choice of the proper scheduling algorithm can significantly improve the overall efficiency of a backup system, and can be considered of great importance in environments where a large number of hosts are involved.

Scheduling algorithms used in the backup domain are different from the ones used in other domains.

In the backup domain, job preemption [19] is often not supported. The majority of backup software products do not provide this feature since the context switching introduced is time consuming especially for systems based on tape libraries. To enable preemption of backup jobs, checkpoints must be created. The complexity required for the creation of checkpoints and its overhead discourage the use of preemption in the backup software domain.

It is also important to note that a backup job usually requires considerable amounts of time and resources. The choice of which job to run among a set of eligible jobs is particularly important in environments where resources are limited.

When dealing with backup systems, the CPU overhead introduced by a complex scheduling algorithm can be often considered negligible.

In this chapter the scheduling algorithms of concern to this thesis are discussed, together with their advantages and disadvantages.
4.1 First-come, first-served scheduling

The first-come first-served (FCFS) strategy is one of the simplest and oldest scheduling algorithms [2], [15], [17], [19].

Usually a single queue is used to gather the jobs that request system resources. Every time a new job is scheduled, it is enlisted at the end of the queue. When resources are available they are allocated to the job in the first position of the queue and the job is removed. Any time resources are released they are assigned to the new head of the queue.

The FCFS algorithm is not preemptive [19] and once a process gets the resources, it runs until completion.

The implementation of this scheduling strategy is usually straightforward and the CPU overhead introduced is minimal, especially if the algorithm is implemented using a FIFO queue.

A major issue with the FCFS algorithm is the convoy effect, which often has a very negative impact on the jobs waiting time.

4.1.1 The convoy effect

The convoy effect [19] occurs when more processes share the same resources. If a long process holds the resources for a very long period while new processes are scheduled, the system load increases significantly, causing additional delays.

The convoy effect is often the cause of unbalanced workload and long waiting time for jobs. To limit the consequences of the convoy effect, shorter jobs should be allowed to run first.
4.2 Priority scheduling

The idea behind the priority scheduling algorithms [19] is simple: each task has a *priority* associated, and this value represents the importance of the job in the system. The scheduler always allocates the resource to the job with the highest importance among all.

Priority scheduling algorithms can rely on *static* or *dynamic* priority strategies.

In *static* priority (or *fixed-priority*) strategies, job priorities never change and remain the same from the start of the job to its completion. Algorithms based on static priorities can suffer of *job starvation* [19].

In *dynamic* priority strategies, job priorities are calculated at runtime and can either increase or decrease according to specific conditions or mechanisms.

Usually, *the lower is the priority value, the higher is the job importance.*

Figure 2 shows a job with static priority (blue line) which has a priority set to 250, and a job whose priority decreases with time (red line).

![Figure 2: Static and dynamic priority](image-url)

4. Background on scheduling algorithms
4.2.1 Starvation and aging

Every time a job is ready to run, but the necessary resources are not available, the process is forced to wait. If a low priority job is continuously overcome by higher priority jobs an *indefinite postponement* may arise and the waiting job has never assigned the needed resources. This situation is known as *starvation*.

Different solutions exists to prevent or avoid job starvation. The simplest solutions is realized by limiting the number of times that a low priority job can be overcome by higher priority jobs. A more sophisticated technique that take into account the possibility of increasing the importance of a job with the passing of time is the *aging* [19] mechanism.

The aging mechanism is used to gradually increase the importance of jobs that are waiting for the necessary resource allocation. The importance of a job is increased at regular intervals of time. Sooner or later the job will succeed in getting the resources as, in the worst case, it will become the most important job in the system.

The price to pay when using the aging mechanism is the *computing overhead* introduced when updating priorities.

Some effort must also be spent to estimate the proper interval of time for aging to take place. If the aging interval is too short then low priority jobs might reach the highest importance very fast, thus reducing the scheduler to a FCFS strategy, if the aging interval is too long the mechanism may result partially ineffective.
4.3 Multilevel queue scheduling

The multilevel queue (MLQ) algorithm [19] is based on the use of multiple queues. Jobs are classified into categories and there is a queue for each category. Also, each queue has a different importance and the algorithm used for selecting the most important job can be different for each queue.

Every time the resources are available, the scheduler selects the first job from the highest priority queue. If the highest priority queue is empty then the less important queues are analyzed until a job is found. The job is then removed from the corresponding queue and the resources are allocated.

Jobs can not be moved between queues and since every job is enlisted in a specific queue, it remains there until completion. The initial positioning of each job in the most suitable queue is therefore of great importance for the overall efficiency of the algorithm.

The parameters that can change from one implementation to another are:

- the number of queues,
- the scheduling algorithm assigned to each queue,
- the algorithm that decides in which queue each job must be placed.
4.4 Multilevel feedback queue scheduling

Multilevel feedback queue (MLFQ) algorithms [10], [15], [19], [21], are extensions to the more general MLQ algorithms.

The main difference relies on the fact that, in MLFQ algorithms, jobs are allowed to move from one queue to another, according to their runtime behavior.

By moving jobs from higher priority queues to lower priority ones, the algorithm guarantees that short jobs and I/O-intensive processes get to the CPU faster. After a quantum of time jobs are decreased in importance and moved to a lower priority queue. Aging can be applied to increase the importance of a job that has been waiting in a queue for too long.

The main difficulty in the application of MLFQ algorithms relies on their complexity. MLFQ algorithms must be tuned appropriately to meet specific requirements and achieve high performance.

When using MLFQ algorithms starvation is just a minor issue, since the possibility of promoting/demoting jobs from one queue to another easily tackles the problem.

MLFQ algorithms analyze the behavior of jobs at runtime. By using statistics (history) and runtime information, jobs can be distinguished into I/O-intensive or CPU-intensive.

Problems with the MLFQ algorithms arise when a process changes its behavior over time and the scheduler is not able to recognize these changes. In these situations the system performance usually decays. Some solutions to these typologies of problems exist, but are not relevant to this treatment.

Generally, MLFQ algorithms achieve better results if compared to other scheduling policies, but the overhead introduced is usually higher. An application of MLFQ scheduling algorithm is implemented in the Linux scheduler [3], [12], [19], [21].
In this chapter different approaches to job scheduling in backup software systems are discussed.

The Bacula internal scheduler is described first, and then an overview of the approaches used in IBM Tivoli Storage Manager, EMC NetWorker, NetBackup and Amanda is reported. We also briefly discuss how scheduling can be achieved by scripting. With the exception of Bacula, whose source code has also been analyzed, most of the information is based on the official documentation for each product.

5.1 Bacula

The Bacula software offers very useful features and has an internal scheduler for the management of jobs over time. The scheduling algorithm adopted is simple and efficient, but somehow limited when compared to other scheduling approaches.

The built-in scheduler relies on the management of a single FCFS queue, where jobs are ordered by time and priority. At runtime the scheduler finds out which job needs to be queued by analyzing the job configurations. Every job scheduled for running within the current or the next hour of the day is enlisted in the jobs_to_run queue.

A priority is assigned to each job. In environments where jobs can run concurrently the priority system adopted in Bacula has some limitations. Bacula does not allow a lower priority job to start if a job with higher priority is already running. For this reason the use of different priorities is inefficient and jobs with lower priorities are always forced to wait for all higher priority jobs to complete, even if the necessary resources are available and concurrency is enabled.
The priority system used in Bacula is static. When the necessary resources to run a job are available, the scheduler extracts the job in the first position of the queue and runs it.

Some other limitations to this approach are:

- A long job can easily cause a convoy effect. When this happens the scheduler keeps on running jobs with a FCFS strategy, basically ignoring the fact that there might be jobs more urgent than others.
- Starvation can arise if a low priority job is constantly overcome by higher priority jobs. The job could be delayed forever waiting for higher priority jobs to complete.
- If the director fails or is shut down, the job schedule is lost. When the system is back on, a new schedule is created and the system does not have any strategy to requeue jobs that were scheduled but not run.

5.2 IBM Tivoli Storage Manager

The IBM Tivoli Storage Manager (TSM) [20] has a central scheduler that takes care of the automatic scheduling of jobs.

A schedule definition is used to specify the type of action to perform and the time for it to execute. The action can be a command, a script, or an operating system command.

Startup windows define the acceptable time interval for a scheduled event to start.

A scheduled action is required to start within the associated startup window, and subsequent events can be defined by specifying a frequency for the action.

TSM allows to associate more clients to the same schedule definition, and also a client can be associated to more than one schedule definition. In this case administrators must make sure that schedule windows do not overlap.

A static priority system is defined, and can be used to give different importance to the schedules.
5.3 EMC NetWorker

*EMC NetWorker* [6], formerly known as Legato NetWorker, schedules jobs using a *timed-based* or a *probe-based* configuration.

The *schedule* resource defines when a backup job must be performed and what is the backup level for each client.

*Timed-based* backups run at a specific day, week or month.

*Probe-based* backups on the other hand are also given a time interval (*start time* and *end time*) where jobs are allowed or not allowed to run. Probe-based backups allow the administrators to run jobs based on *events* or *conditions*. These events or conditions are enabled by *scripting* techniques, so that, when they are are met a specific backup operation is performed.

EMC NetWorker uses the concept of *groups*. Clients are grouped together and when the start time for a group of backups arrives, all associated clients are scheduled for running. By using *groups* the administrators can balance the system workload simply by differentiating jobs into different sets, different schedules, and furthermore permits to sort data into specific volumes and pools.

Administrators must define the *start time* for each group according to the estimated time required by each backup job to complete.

A priority can be set for each client. The backup order inside the same group depends on these values, if the *priority* is not set the backup order is random.
5.4 NetBackup

NetBackup [14] lets administrators define time windows inside which jobs can be started.

The schedule resource contains the information related to the job planned start and the backup level to perform.

Jobs can be either set for running in fixed days, weeks or months, or can be set for running at regular intervals of time. By setting a job frequency, the amount of time between the successful completion of a scheduled job and its next instance is calculated accordingly.

If more than one backup is scheduled for running at the same time, jobs with the lower frequency (i.e. longer periods between backups) are assigned higher priorities. If the frequency is the same and jobs are all within their time windows, then they are run in alphabetical order.

When multiple schedules are defined, NetBackup calculates the due time for each schedule and selects the schedule with the earliest due time. The due time depends on the last backup performed. If jobs are frequency-based jobs, then the due time is calculated according to the frequency attribute, that is: Due time = Last backup data + Frequency.

If jobs are scheduled to run at fixed days, the due time corresponds to the time planned for the next job instance.

In NetBackup, “an higher priority does not guarantee that a job receives resources before a job with lower priority”[14]. This means that, in some cases, lower priority jobs may run before higher priority ones if some specifics conditions are met. These conditions may be related to the status of the tape storage system, evaluation cycles and group multiplexing[14].
5.5 Amanda

Amanda [1], acronym for Advanced Maryland Automatic Network Disk Archiver, is an open source backup software distributed over the BSD license. Depending on the network and storage availability, the internal scheduler determines the optimal backup level to perform for each client. When resources are overwhelmed, backups are delayed or even canceled if necessary.

The maximum time between full backups is called a dump cycle. For any dump cycle, Amanda elaborates an optimal balance of full and incremental backups of all clients. Amanda uses the following information to elaborate an optimal plan:

- The total amount of data to be backed up,
- The maximum dump cycle specified,
- The available storage resources for each backup run.

An estimation phase precedes every backup run: every client runs (locally) a procedure to identify the changes occurred in the file set and the size of these modifications. Although this phase is time consuming, it is necessary before performing the planning phase, which takes care of estimating the optimal combination of full and incremental backups for all clients.

5.6 Scripting

Almost all backup software products have scripting capabilities. Scripting techniques can be used for the automatic scheduling of jobs. Rdiff-backup [16] only achieves scheduling by scripting and allows for the integration with external job schedulers (Cron [5]). Other features are also obtained by scripting (backups triggered by external conditions, as in EMC NetWorker), and the majority of backup products also allows for the execution of scripts before and after running a backup job.
5.7 General considerations

The main reason why backup software products rarely deal with sophisticated scheduling techniques is probably due to the fact that simple and intuitive strategies such as the first-come-first-served or the priority scheduling algorithms often provide all the necessary capabilities for small/medium size organizations with minimum effort. Furthermore, the schedules created by these algorithms are predictable and this makes the overall backup planning activity easier for administrators.

Static priority strategies are often used in order to give different relevance to scheduled jobs, but the issues that may arise in worst case scenarios are not dealt explicitly.

In general, the approaches used for the scheduling of backup jobs by software products in the backup domain are very similar and present only few variations and peculiarities.

Common characteristics are:

- Simple scheduling algorithm
- Static priority strategy
- Grouping of jobs
- Time windows
- Workload balancing capabilities
- Fixed-time and frequency-based job schedules

Each product possesses its own peculiarities.

Amanda is the only one providing an “automatic” estimation of the backup level for each job, and automatically attempting to balance the system workload.

NetBackup provides a different priority approach where jobs with lower priorities can, under certain circumstances, overcome higher priority jobs.

NetWorker gives the possibility to check for specific runtime conditions before enabling specific backups. Furthermore, it suggest the combined use of time windows and groups to
manually achieve workload balance.

TSM gives the possibility to associates more clients to the same scheduler, or to associate more schedules to a single client, so to create more complex scheduling behaviors.

Bacula offers a built-in scheduler which adopts a static priority system and allows for jobs concurrency. In the following sections we introduce a new implementation for the Bacula scheduler and discuss how this new scheduler is able to:

- prevent job starvation,
- limit the effects caused by convoy effects,
- recover a lost schedule after a system interruption.
In this chapter a new scheduler for the Bacula software is introduced. It is based on an experimental scheduling algorithm that could be adopted by different backup software products.

The proposed algorithm reflects most of the characteristics of the MLFQ scheduling algorithm category. Jobs are moved among queues either depending on their runtime status or their dynamic priority values.

In this treatment we will assume the following:

*the lower is the dynamic priority number, the higher is a job relevance.*

### 6.1 Job configuration

Bacula provides a wide set of *directives* for the configuration of a job. This set of directives is enriched with new attributes, two of which are mandatory. The *starting priority* and the *aging* directives must be defined for each job while the others are optional. All directives must have positive values.

*Starting Priority*

The *starting priority* represents the initial importance of the job.
**Aging**

The *aging* directive defines how fast a job gains importance as time passes. If the scheduler runs the aging procedure and the job is waiting for the allocation of resources, the job dynamic priority is decreased of the value specified by this directive.

**Early Start**

The Bacula *schedule* directive (section 3.4) defines when a job must be scheduled for running. Under specific circumstances, the *early start* directive is used by the scheduler to promote jobs by running them ahead of their planned start time.

**Local Highest Priority**

This directive defines a limit to the lowest dynamic priority a job can reach due to the aging mechanism. By default the scheduler assumes that the job can reach the lowest dynamic priority defined in the scheduler configuration (section 6.4). The use of this directive may cause job starvation.

**Periodic, Period, Reference**

The *periodic* directive is a boolean value. If it is unset, then the job is regularly scheduled when its planned start time comes. If the periodic directive is set, the scheduler computes the next job planned start time according to the *period* and the *reference* directives.

The *period* directive defines the interval of time that must elapse between the last successful job instance and the start of the next one.

The *reference* directive is used by the scheduler to determine if the interval of time defined in the *period* directive must be calculated with respect to the *planned start time*, the *actual start time* or the *completion time* of the last job instance performed.
Note that planned start time and the actual start time of a backup job might differ. The former represents the time that the scheduler takes as reference to enqueue the job in the appropriate queue, the latter corresponds to the real job start time.

**On Failure, Failure Delay**

The onFailure directive defines how many times a job can be rescheduled before being permanently canceled.

The FailureDelay directive defines the amount of time that must elapse between the last job failure and the start time of the next attempt.

**Time Window, Time Window Type, Penalty, Block Aging**

The time window directive defines an interval of time in which a job must be treated in a particular manner.

The time window type defines how a job influenced by a time window must be treated. If type is set to blocked, then the scheduler inhibits the job from starting even if the resources are available. If type is set to penalty the dynamic priority of the job is temporarily increased of a value equal to the one defined by the penalty directive.

Aging can be disabled in time windows of type blocked by setting the blockAging directive.

Note that a time window can only prevent a job from starting, but can never stop a running job.

Figure 3 shows how the new directives affect a job dynamic priority over time.
The start priority defines the initial value of the job dynamic priority, that is 250 in the example. For a period of 4 hours the job is in its early start interval, meaning that the scheduler could optionally run the job ahead of the expected start time.

When the job planned start time arrives (t = 4), if the job is not run because of resources unavailability then its dynamic priority is periodically decreased.

The local highest priority directive (LHP), set to 60 (t = 23), stops the aging mechanism and avoids that the dynamic priority reaches the global lowest value, which in the example is supposed to be 0.

Two examples of time windows are also reported. The first is a blocked time window (from t=8 to t=11): during this interval the scheduler will not start the job even if there are available resources. The second is a penalty time window (from t=13 to t=17): the dynamic priority of the job is increased of its penalty value, set to 20. For the whole duration of this time window, the importance of the job is decreased.

Figure 3: New job behaviors

The start priority defines the initial value of the job dynamic priority, that is 250 in the example. For a period of 4 hours the job is in its early start interval, meaning that the scheduler could optionally run the job ahead of the expected start time.

When the job planned start time arrives (t = 4), if the job is not run because of resources unavailability then its dynamic priority is periodically decreased.

The local highest priority directive (LHP), set to 60 (t = 23), stops the aging mechanism and avoids that the dynamic priority reaches the global lowest value, which in the example is supposed to be 0.

Two examples of time windows are also reported. The first is a blocked time window (from t=8 to t=11): during this interval the scheduler will not start the job even if there are available resources. The second is a penalty time window (from t=13 to t=17): the dynamic priority of the job is increased of its penalty value, set to 20. For the whole duration of this time window, the importance of the job is decreased.
6.2 Scheduling strategy

In this section, the queue design and the job categories are introduced. The scheduling algorithm is described in detail in section 6.2.2.

Appendix B reports the pseudo-code of each of the new scheduling algorithm functions.

6.2.1 Queue design

Multiple queues are used. Each queue is associated to a single job status, and vice versa. At runtime a job can be in one of the following status: ready, early start, waiting, blocked, penalized, high_priority, local_high_priority, failed, canceled, inactive. Jobs can be moved from one queue to another but they can never be in two different queues at the same time.

The defined queues are listed below.

**Ready Queue**

The *ready* queue contains the jobs that are ready for running as soon as the needed resources become available. This queue is not ordered and the most important job is the one with the lowest dynamic priority value among all.

**Early_start Queue**

The *early_start* queue contains jobs that can be scheduled ahead of their planned start time if the system is idle. The most important job is the one with the lowest dynamic priority value among the whole set. Jobs are ordered by planned start time.
**Waiting Queue**

The *waiting* queue contains all the jobs that are waiting for their planned start time to come. This queue is ordered by planned start time.

**Blocked Queue**

If a job is inside a time window of type *blocked* then the job is enqueued in the *blocked* queue for the whole duration of the time window. None of the jobs in this queue can be scheduled for running. This queue is not ordered.

**Penalized Queue**

The *penalized* queue contains the jobs that are inside a time window of type *penalty*. These jobs can be run, but their *dynamic priority* values are *temporarily* increased of a value defined in the *penalty* directive. This queue is not ordered.

**High Priority Queue**

The *high priority* queue gathers the jobs that have reached the *lowest dynamic priority* value and therefore possess the highest priority in the system. Jobs in this queue are selected with a FCFS algorithm.

**Local High Priority Queue**

Only jobs that have been provided with a *local highest priority* directive can be enlisted in this queue. If a job dynamic priority reaches the value defined in the *local highest priority* directive, then the job is moved to this queue. This queue is kept ordered by dynamic priority values.
**Failed Queue**

The *failed* queue maintains the jobs that did not complete successfully the last run. When a job fails, it is enlisted in this queue.

**Canceled Queue**

The *canceled* queue contains the jobs that have failed more times than allowed in the respective job configuration. Jobs in this queue are permanently inactive and, if necessary, can be restored manually by administrators.

**Inactive Queue**

The *inactive* queue contains the jobs that are not going to be scheduled “soon”. This queue is not ordered.
6.2.2 Scheduling algorithm

The additional directives and the queues described so far are necessary for the application of the proposed scheduling algorithm.

A *dynamic priority* value is associated to each job and represents the relevance of each job in the system at runtime. Due to an *aging mechanism*, this value can change as time passes, and for this reason is referred as *dynamic*.

If a job is ready for running but the required resources are not available, the *aging mechanism* makes sure that *dynamic priority* value is periodically decreased.

Each job that is planned for running is enlisted in one of the queues.

A first screening of jobs is required for separating *active* jobs from *inactive* ones. Active jobs are those that the scheduler monitors continuously and are expected to run within a “short” interval of time. The remaining jobs are defined inactive, and therefore enqueued in the *inactive* queue. When the screening must be performed depends on the specific setup. In the scheduler we propose, the screening is performed once a day, at midnight, and the jobs that are not planned for starting within the day are considered inactive.

The *active* jobs are distributed over the remaining queues, depending on their *status*. Job status changes, the job is moved to the queue associated to the new status.

If a job is waiting for its planned start time it is placed in the *waiting* queue unless the *early start* directive was defined for the job. In this case the job is placed in the *early_start* queue. When the planned start time for a job arrives, the job is moved into the *ready* queue, unless differently defined.

Every time a job reaches its lowest dynamic priority it is moved either to the *highest priority* queue or to the *local high priority* queue if a *local highest priority* was set. Penalty time windows are disabled for jobs in these queues, while *blocked* time windows, if defined, apply normally.

Any time a job enters a time window, it is moved to either the *blocked* queue or the *penalty*
queue, depending on the time window type. The only exception is represented by the high priority queue in which penalty time windows are disabled and blocked time windows are applied without moving the job to the blocked queue. This guarantees that the enlisting order of jobs in the high priority queue is respected and prevents job starvation.

If a job instance fails it is moved to the failed queue where it is rescheduled in accordance with the onFailure and FailureDelay directives. If the job is allowed to rerun the job is moved to the most appropriate queue, otherwise it is moved to the canceled queue where it is considered permanently failed.

When the necessary resources are available and a job can be run, the following strategy applies for the election of the most important job:

- If the highest priority queue contains jobs, then jobs are extracted with a FCFS algorithm and resources are allocated to the first non-blocked job in the queue.

- If any of the ready, the penalized or the local high priority queues contains jobs, then the job with the lowest dynamic priority among all is selected for running.

- If none among the ready, the penalized, the highest priority and the local high priority queues have jobs ready for running, then the job in the early_start queue with the the lowest dynamic priority value is run.

Every time a job is selected for running, it is removed from the corresponding queue.

If the system allows more jobs to run concurrently, the procedure for finding a job to run can be repeated as many times as the number of concurrent jobs permitted.

When a job is completed, its configuration is rechecked so to determine if the job needs to be requeued. If so, the next planned start time is computed: in case of periodic jobs, the period and the reference directives are used together with the information related to the last successful backup; if the job is not periodic, the next scheduled instance is used. In any case the new planned start time is analyzed and the job enlisted in the most appropriate queue.
6.2.3 Aging process

The aging mechanism is periodically applied.

A time unit is a time interval defined in the scheduler configuration which is the basic unit for
the aging process to be applied. All jobs that are waiting for resource allocation are affected,
and the dynamic priority value of each job is decreased of a quantity equal to the aging
directive specified in every job configuration.

Aging is not applied in the following cases:

• The job is either in the high priority or in the local high priority queue. There is no
  reason to apply aging since the priority can not be increased.

• The job is in the early_start queue. Due to the fact that the job is expected to run only
  if the system is idle, the aging process is disabled.

• The job is in the blocked queue and the blockAging directive is set.

• The job is in the inactive queue.

The aging mechanism prevents the starvation problem. Sooner or later, jobs that have been
waiting for a very long time will enter in the high priority queue and, since a FCFS strategy
applies to this queue, every job will be guaranteed the needed resources.

Starvation can arise if the local highest priority directive is used. This feature is used to
specify particular job behaviors, especially background jobs (see section 7.1), and is useful in
system with irregular workload. Administrators must consider the possibility of job starvation
when using the local highest priority directive.
6.3 Schedule recovery feature

As Bacula runs, it collects and stores data about completed jobs in the catalog database.

The reuse of the information contained in the catalog allows to define procedures for the recovery of the schedule in case of system anomalies. If the backup server is powered off (e.g. in case of a planned maintenance) or the director process is interrupted for any reason, the Bacula scheduler loses both the schedule and the dynamic priority values associated to each job. This schedule can be recalculated with a good approximation by analyzing both the jobs history stored in the catalog and the job directives.

In appendix B the pseudo-code of the schedule recovery procedure is reported.

The scheduler first verifies if a job was planned for running, then a new estimation of the dynamic priorities is performed and finally the job is queued into the appropriate list. The scheduler must carefully consider the characteristics of jobs, especially when dealing with periodic jobs. Multiple instances of the same job must never be replicated.

The schedule obtainable by the recovery feature is an “approximation” in the sense that the lost schedule can not be recreated exactly. The job configuration is analyzed again and the high priority queue and the local high priority queue can be altered if compared to the original plan. This is due to the fact that the parsing of the configuration is an iterative procedure and produces a plan which is different from the one obtained at runtime which is influenced by the aging process.

Please note that if the backup system downtime has been very long, the recalculated schedule may contain a significant number of jobs at their highest priority.

If the administrator wants to use the scheduler recovery procedure, the schedule recovery directive must be defined in the scheduler configuration.
6.4 Scheduler configuration

Every scheduler based on a MLFQ scheduling algorithm requires additional configuration to achieve the best performance. Also in this case some effort is required for an appropriate scheduler configuration.

Depending on the characteristics of the environment where the scheduler must be deployed, administrators might want to override the default values of the following parameters:

- the interval of time for the scheduler to run its main procedure,
- the *aging time unit*, which is the interval of time for the aging process to take place,
- the highest and the lowest dynamic priority values range,
- limits for job configuration parameters: the maximum aging per unit, the maximum number of job failures, etc.

The recovery procedure must be explicitly enabled before launching the scheduler.
7. Analysis of the MLFQ scheduler

In this chapter a discussion on the new scheduler and its scheduling algorithm is presented.

Section 7.1 provides an analysis of new job behaviors which can be defined using the new directives. Section 7.2 provides a complexity estimation of the procedures performed by the original and the new schedulers. Section 7.3 compares the original scheduler with the new one using a case study. Section 7.4 discusses guidelines to consider when setting up a backup plan using the new scheduler, and finally section 7.5 discusses a preliminary analysis on the behavior of the new scheduler after its deployment in one of the LNGS production chain.

7.1 General analysis

The set of new job directives introduced in section 6.1 allows administrators to specify different job behaviors.

EARLY START

The early start feature can be used to improve the resources utilization under conditions of low system workload. When the resources are idle, the scheduler looks for jobs that can be “promoted” with an early start. If a job is found, the scheduler allocates the resources and runs the job. By starting a job earlier than planned, the resources will also return free earlier.

Since backup jobs often require considerable amounts of time, this feature is useful in environments where system workloads are not always balanced over time.

Even though this feature reduces the idle time of resources, delay is introduced for jobs which become ready as soon as an early start job is launched. Consider the following example.
Example 1: Early start jobs

Suppose that three important jobs, job 1, job 2 and job 3, are scheduled to run respectively at 00:00, 1:30 and 3:00.

The administrator expects these jobs to be scheduled as shown in figure 4.

![Figure 4: Example 1: High priority jobs planned start](image)

Now suppose a new job, job 4, to have a low priority, a planned start time set at 5:00, an early start directive of 4 hours, and a estimated duration of 1:30 hour. At time 1:00, the scheduler detects the resource availability and the possibility to promote job 4. Job 4 is then run at 1:00. At time 3:00 the resource returns available again for job 2 and job 3, but job 4 has introduced an additional delay for both of them (figure 5).

![Figure 5: Example 1: Jobs delayed by a lower priority job](image)
PERIODIC JOBS

With the *periodic*, *period* and *reference* directives administrators can define jobs that run at regular intervals of time.

Suppose that two instances of the same job are scheduled for running not too far from each other. If the first instance is delayed, the second instance may run too close to the first. Having two instances of the same job run too close to each other is often a waste of resources.

Periodic jobs can be defined to overcome this limitation. The advantage of using periodic jobs becomes relevant when there are high chances of long delays. Periodic jobs guarantees that the next job instances are scheduled after proper time intervals.

Figure 6 shows an hypothetical periodic job behavior. Job 1 is always expected to run with a periodicity of one hour after the completion of the latest instance. The orange rectangle represents a different job that delays the second instance of job 1. Its third instance will be scheduled, as expected, one hour after the completion of the second. If the job schedule were defined on a fixed-time basis, then the second and the third instances would have run too close to each other.

![Figure 6: Periodic job](image)

7. Analysis of the MLFQ scheduler
TIME WINDOWS

Time windows of type *blocked* specify intervals of time where jobs must not be started. Administrators can define time windows for single jobs, for sets of jobs, or as default for every job in the system.

If time windows are used for a single job, they can express specific conditions related to the client machine, such as the *impossibility* to reach the host in specific hours or days, or hours related to peak activity and therefore *not recommended* for backups.

If time windows are defined for groups of jobs, interesting system behaviors can be enabled. Administrators can reserve specific interval of time for administrative purposes (specific backups or data verification jobs) by instructing the scheduler to block the automatic start of backups.

The use of time windows also compacts jobs together, so that run one after the other as soon as the time windows end. This result in minimum resources idle time but can produce consistent delays for high priority jobs.

*Penalized* time windows have a direct impact on the importance of one or more jobs. They can be used to decrease the importance of jobs in some hours of the day or in some specific days. This feature becomes useful when the system is known to have workload peaks (for example if many jobs are planned for starting in a specific day or week) and the administrator wants to reduce the importance of some groups of jobs (jobs that have been run recently for example). Decreasing the priority of specific jobs may allow other categories of jobs to run before, still guaranteeing the penalized jobs to run if their importance become relevant or the system resources are under-utilized. Penalized time windows are more effective on the long term planning of backups.

7. Analysis of the MLFQ scheduler
AGING

The *aging* directive is a powerful instrument. This value influences the importance of each job in the system and defines its behavior when time passes.

Example 2 reports an illustrative example of how aging influence the importance of the jobs over time.

**Example 2. Jobs with different aging values**

Suppose that job 1 planned start time comes sooner than job 2 planned start time. Suppose also that job 2 needs to be run more often than job 1 and that job 1 is supposed to reach its highest importance in a longer time than job 2 (figure 7).

![Figure 7: Example 2: Job with different aging values](image)

The different aging values (higher value for job 2) makes sure that job 2 becomes more important than job 1 after a suitable interval of time.

Under these circumstances, job 1 is run before job 2 only if the resources become available before their dynamic priorities reach the same value, which happens at t=16. After that, the
scheduler prefers to run job 2 instead of job 1.

The situation described in the example can represent jobs of different typologies, such as full backups (job 1) and incremental backups (job 2).

**LOCAL HIGHEST PRIORITY**

The *local highest priority* directive allows administrators to schedule jobs that must not become too important in the backup plan.

If the directive is applied to sets of jobs, it can be useful for workload balancing. These jobs will only run when the resources are available and no higher priority jobs are queued.

If appropriately defined, jobs in this category can be considered as *background* jobs: they do not always compete for resource allocation, but they wait for the first unused time slot. This feature can generate job starvation if used inappropriately.

**MIXING DIRECTIVES**

Although each feature used on its own gives administrators the possibility to define complex behaviors, it is from the combination of them that the greatest advantages are achieved.

Mixing directives can result in really interesting behaviors. For example, a job could be set to be *periodic*, with an *early start* and with *blocking* and *penalty time windows*. The *aging* directive can be set very high or very low, and the *local highest priority* can also set to limit the importance of the job.
7.2 Complexity analysis

The new scheduling algorithm belongs to the MLFQ category.

The benefits brought by a scheduler based on the new algorithm are relevant in the backup domain but its complexity must also be considered.

The management of the queues, the operations required to keep queues ordered, the aging mechanism are at the basis of the computational overhead introduced by the new scheduler.

The time required by a backup operation is orders of magnitude greater than the time required to run the scheduling algorithm. Also, when performing a backup in Bacula, the director launches most of the backups “asynchronously” and does not continuously monitors the client. Under these conditions, the director does not perform any relevant computation beside updating the catalog. In the scheduler proposed, the overhead introduced is negligible, but some operations of the scheduling algorithm must be analyzed, since they are linear in complexity.

Also the set of Bacula data structures helps the scheduler to remain efficient. Most of the checks performed during the procedure are not performed on integer data structures, but on bits.

The screening procedure and the procedure to discover if time windows apply to jobs are based on bit comparison and ensure fast computation. This is possible because Bacula uses bit masks to define the schedule of jobs [18].
7.2.1 Original scheduler complexity

The operations performed by the original Bacula scheduler are: insertion of a job, selection of the job to run, and extraction of a job from the jobs_to_run queue. Appendix A contains the pseudo-code for the original scheduler.

JOB INSERTION

When a job must be inserted into the jobs_to_run queue, the scheduler checks which is the most appropriate position for the job. In the worst case, the scheduler scans the whole queue.

The cost of inserting a job in the queue is $O(n)$, where $n$ is the number of jobs in the queue.

JOB EXTRACTION

The cost of extracting the job from the queue is $O(1)$.

SELECTION OF THE JOB TO RUN

When resources are available, the scheduler selects the job in the first position of the jobs_to_run queue.

The cost of finding the job to run is $O(1)$.
7.2.2 New scheduler complexity

The operations concerning the new scheduler and their complexities are reported below. Appendix B reports the pseudo-code of the new scheduler.

JOB INSERTION - ORDERED QUEUE

The insertion of a job into any of the ordered queues is performed by appending the job in the most appropriate position. In the worst case, the insertion of a job requires the whole list to be analyzed, therefore the insertion of a job into an ordered queue is is $O(m)$, where $m$ is the number of jobs in the queue.

JOB INSERTION - UNORDERED QUEUE

The insertion of a job into any of the unordered queues is performed by appending the job at the end of a queue. Each queue has a pointer to its last element, therefore the cost of inserting a job into an unordered queue is $O(1)$.

JOB EXTRACTION

The cost of extracting a job from any queue is $O(1)$.

MOVING A JOB TO A DIFFERENT QUEUE

The cost of moving a job from one queue to another is the sum of the operations of job extraction and job insertion. It is $O(1)$ for insertions into an unordered queue and is $O(m)$ for insertions into any ordered queue, where $m$ is the number of jobs in the ordered queue.
ELECTION OF THE JOB TO RUN

When selecting the most important job to run, the scheduler first checks for non blocked jobs in the high priority queue.

In the worst case, all the jobs in the high priority queue are blocked and the scheduler must parse the entire queue. Assuming \( m \) the number of blocked jobs into the high priority queue, the cost of this operation is \( O(m) \).

This cost must be added to the cost of the following three possible scenarios:

1. **JOBS IN THE LOCAL HIGH PRIORITY, READY, OR PENALIZED QUEUES**

   If the local high priority queue, the ready queue and the penalized queue have jobs, then the scheduler looks for the most important job among these queues. The cost of this operation is \( O(m+n+p) \), where \( m, n \) and \( p \) are respectively the number of jobs in the local high priority, the ready and the penalized queues.

2. **NO JOBS IN THE LOCAL HIGH PRIORITY, READY, OR PENALIZED QUEUES**

   If the local high priority queue, the ready queue and the penalized queue have no jobs, then the scheduler looks for the most important job in the early_start queue, which in the worst case is the one in the last position. The cost of this operation is \( O(m) \), where \( m \) is the number of jobs in the early_start queue.

3. **NO JOB TO RUN IN THE LOCAL HIGH PRIORITY, READY, PENALIZED OR EARLY_START QUEUE**

   In this case, all queues are empty, and the complexity of this operation is \( O(1) \).
AGING CYCLE

The cost of applying the aging mechanism to a single job is constant independently from the number of time units detected by the scheduler.

Aging is basically applied only to jobs in the ready and the penalized queues. Its cost is therefore $O(m+n)$, where $m$ is the number of jobs in the ready queue and $n$ is the number of jobs in the penalized queue.

JOB SCREENING

The cost related to the screening of jobs into the active and the inactive category is $O(m)$, where $m$ is the total number of jobs in the system.

7.2.3 Scheduler complexity comparison

In section 7.2.1 and section 7.2.2 the complexity of the operations for both the original and the new the scheduler are reported. Appendix A and Appendix B contain the relative pseudo-codes.

The scheduling complexity is similar when dealing with job insertions, but, in the worst case, the election of the best job to run is achieved in a linear time by the new scheduler, while the original scheduler is able to elaborate it in constant time. The procedure for finding the best job to run is now more complex, but is essential to the new scheduler.

In the original scheduler, the jobs_to_run queue contains the jobs have to be scheduled for running in the same hour and the next. This queue is updated when new jobs are detected.

In the new scheduler, the initial job screening procedure allows to significantly reduce the number of jobs to monitor. With this screening the jobs are actually separated into two main categories: the jobs that will run within the day (including those that are scheduled for the
following days, but with a possible *early start* within the day) and those that will not.

The *active* jobs are analyzed and dispatched into the most appropriate queue, while the remaining ones are moved into the *inactive* queue. The workload for the new scheduler is usually heavier at the beginning of each day.

According to its configuration, the new scheduler periodically applies the aging mechanism. Aging is not applied to jobs in the *inactive* queue nor to jobs in the *early start* or *blocked* queues. For obvious reasons, jobs in the *highest priority* and *local priority* queues are not affected by aging as well. The *ready* queue and the *penalized* queue are, on the other hand, periodically updated.

The scheduler must also move jobs from one queue to another when their status change at runtime. If a job dynamic priority reaches its lowest value, if time triggers a change (waiting to ready queue, early_start to waiting or ready, etc.) or if a time window is applied, the job status is updated.

Time windows for blocks and penalties are defined on an hour basis granularity. This is a choice due to the *schedule* data structure implemented in Bacula.

Let $N$ be the total number of jobs to be scheduled in the system. Let $m$ be the number of jobs detected to be inactive. Let $n$ be $(N-m)$ the number of jobs distributed over the remaining queue. The update procedure that periodically takes place, costs in the worst case $O(n)$.

The complexity of the main procedure (*Appendix B: main_procedure*) is $O(n)$ with respect to $n$, defined as the number of jobs distributed over the active queues.

7. Analysis of the MLFQ scheduler
7.3 Scheduler comparison

The original Bacula scheduler does not allow low priority jobs to start if high priority jobs are already running. In environments where job concurrency is possible, the only solution to prevent this limitation is by setting all job priorities to the same value. This allows administrators to make sure that resources are used concurrently in case of heavy system loads. If jobs are not distinguished by priorities a basic FCFS strategy is applied by the original scheduler.

In this section the FCFS strategy and the static priority system offered by the original scheduler are discussed and compared to the behavior of the new scheduler.

We define the waiting time for a job as the amount of time that the job has spent waiting in either one of the queues that could schedule a job for running (all queues, except the inactive, the blocked and the canceled queue).

Example 3 reports a case study for the comparison of the two schedulers.

Example 3

Thirteen different jobs of various durations are scheduled for running in 24 hours. Jobs have different priorities and different starting times. We assume that the system is started at 19:00 and that there are no jobs either running nor holding the resources. Also, for simplicity, we assume that jobs have five different priority levels: very low (VL), low (L), medium (M), high (H), very high (VH).

With the exception of job 9, which is considered particularly important, jobs have been assigned priorities according to their duration. The shorter the job is expected to be, the higher is its priority. The aging directive is assumed to affect all jobs the same way.

Table 1 shows the planned start time, the expected duration and the priority associated to each job, while table 2 shows the real starting time of each job according to the backup plans elaborated by the original and the new schedulers.
<table>
<thead>
<tr>
<th>Job</th>
<th>Scheduled time</th>
<th>Duration</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19:00</td>
<td>4</td>
<td>VL</td>
</tr>
<tr>
<td>2</td>
<td>20:00</td>
<td>2</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>20:00</td>
<td>1:30</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>00:30</td>
<td>3:30</td>
<td>VL</td>
</tr>
<tr>
<td>5</td>
<td>02:00</td>
<td>2</td>
<td>L</td>
</tr>
<tr>
<td>6</td>
<td>02:00</td>
<td>3:30</td>
<td>VL</td>
</tr>
<tr>
<td>7</td>
<td>07:00</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>8</td>
<td>12:00</td>
<td>0:30</td>
<td>M</td>
</tr>
<tr>
<td>9</td>
<td>12:00</td>
<td>1</td>
<td>VH</td>
</tr>
<tr>
<td>10</td>
<td>12:30</td>
<td>3:30</td>
<td>VL</td>
</tr>
<tr>
<td>11</td>
<td>12:50</td>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>12</td>
<td>13:00</td>
<td>0:30</td>
<td>H</td>
</tr>
<tr>
<td>13</td>
<td>16:00</td>
<td>2</td>
<td>L</td>
</tr>
</tbody>
</table>

*Table 1: Example 3: Jobs planned start, duration and priority*

<table>
<thead>
<tr>
<th>Job</th>
<th>Start (original plan)</th>
<th>Start (new plan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.00</td>
<td>19.00</td>
</tr>
<tr>
<td>2</td>
<td>23.00</td>
<td>0.30</td>
</tr>
<tr>
<td>3</td>
<td>1.00</td>
<td>23.00</td>
</tr>
<tr>
<td>4</td>
<td>2.30</td>
<td>4.30</td>
</tr>
<tr>
<td>5</td>
<td>6.00</td>
<td>2.30</td>
</tr>
<tr>
<td>6</td>
<td>8.00</td>
<td>9.00</td>
</tr>
<tr>
<td>7</td>
<td>11.30</td>
<td>8.00</td>
</tr>
<tr>
<td>8</td>
<td>12.30</td>
<td>15.00</td>
</tr>
<tr>
<td>9</td>
<td>13.00</td>
<td>12.30</td>
</tr>
<tr>
<td>10</td>
<td>14.00</td>
<td>15.30</td>
</tr>
<tr>
<td>11</td>
<td>17.30</td>
<td>13.30</td>
</tr>
<tr>
<td>12</td>
<td>18.30</td>
<td>14.30</td>
</tr>
<tr>
<td>13</td>
<td>19.00</td>
<td>19.00</td>
</tr>
</tbody>
</table>

*Table 2: Example 3: Jobs start time*
Figure 8 and figure 9 show the real job start time and the time spent waiting by each job. Respectively, they show the original scheduler plan and the new scheduler plan.

**Figure 8: Example 3: Original scheduler plan**

**Figure 9: Example 3: New scheduler plan**

7. Analysis of the MLFQ scheduler
From figure 8 and figure 9 it is possible to see how shorter jobs precede longer jobs in the new plan (in particular: job 3, 5, 7, 11, 12). Job 9 gains positions in the schedule plan because of its high priority. It runs as soon as resources become available.

Job 8 is not only delayed by job 9, but also by job 11 and job 12 because of its low priority.

Jobs 4 and 6 have the same priority and their order of execution is respected.

Job 6 simulates the convoy effect and forces jobs 8, 9 and 10 to wait for the availability of resources.

The job waiting times are reported in table 3 and shown in figure 10.

In the new scheduler, if shorter jobs are assigned higher priorities, the average waiting time is reduced. This is a consequence of the fact that higher priority jobs are not delayed as much as in the original schedule, and therefore they complete faster.

This situation is evident if jobs 2 and 3 are compared. Same reasoning applies for jobs 5 and 6. An exception are jobs 8 and 9. Job 9 has an higher priority than job 8 but also an longer duration, and this causes the average waiting time to grow. This is due to the fact that job 9 was set for running as soon as possible (VH priority).

The total waiting time for jobs in the original plan is about 40 hours, while in the new plan is reduced to about 31 hours.

In average, each job waits for 187 minutes in the original and for 146 minutes in the new plan.

High priority jobs (jobs 9, 11 and 12) have their waiting times consistently reduced from a total of 670 minutes to a total of just 160 minutes. On the other hand, jobs with very low priority (jobs 1, 4, 6, 10) have, in the original plan, an average waiting time of 142.5 minutes each, while in the new plan it grows to 210 minutes each.
<table>
<thead>
<tr>
<th>Job</th>
<th>Planned Start</th>
<th>Waiting Time (original plan)</th>
<th>Waiting Time (new plan)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19:00</td>
<td>0:00</td>
<td>0:00</td>
</tr>
<tr>
<td>2</td>
<td>20:00</td>
<td>3:00</td>
<td>4:30</td>
</tr>
<tr>
<td>3</td>
<td>20:00</td>
<td>5:00</td>
<td>3:00</td>
</tr>
<tr>
<td>4</td>
<td>0:30</td>
<td>2:00</td>
<td>4:00</td>
</tr>
<tr>
<td>5</td>
<td>2:00</td>
<td>4:00</td>
<td>0:30</td>
</tr>
<tr>
<td>6</td>
<td>2:00</td>
<td>6:00</td>
<td>7:00</td>
</tr>
<tr>
<td>7</td>
<td>7:00</td>
<td>4:30</td>
<td>1:00</td>
</tr>
<tr>
<td>8</td>
<td>12:00</td>
<td>0:30</td>
<td>3:00</td>
</tr>
<tr>
<td>9</td>
<td>12:00</td>
<td>1:00</td>
<td>0:30</td>
</tr>
<tr>
<td>10</td>
<td>12:30</td>
<td>1:30</td>
<td>3:00</td>
</tr>
<tr>
<td>11</td>
<td>12:50</td>
<td>4:40</td>
<td>0:40</td>
</tr>
<tr>
<td>12</td>
<td>13:00</td>
<td>5:30</td>
<td>1:30</td>
</tr>
<tr>
<td>13</td>
<td>16:00</td>
<td>3:00</td>
<td>0:30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40:40</td>
<td>31:40</td>
</tr>
</tbody>
</table>

Table 3: Example 3: Job waiting times

Figure 10: Example 3: Waiting time comparison

7. Analysis of the MLFQ scheduler
Resource usage optimization is one of the main goals of the new scheduler. The scheduler allocates the available resources to all the jobs that are ready to run, never holding resources unused.

The new scheduler could simulate the original scheduler priority system if the starting priority and the local high priority of each job were to be set to the same value, but job concurrency would be enabled in any case.

### 7.4 Tuning guidelines

All the schedulers based on a MLFQ algorithm have an inner complexity, that is the proper tuning of the scheduler parameters.

In the scheduler proposed, two fundamental parameters are:

- the time unit interval for the application of the aging mechanism, and
- the dynamic priority range (lowest and highest values for the dynamic priority).

The values assigned to these parameters have a direct impact on the definition of the starting priority and the aging directives of each job.

These parameters depend on the number and the characteristics of the clients to back up.

The granularity achievable by the scheduler is really high, since it is based on unsigned integer data structures. The dynamic priority values can range from 0 to 2^32-1, with 0 being the lowest value.

The highest dynamic priority value can be configured with a lower value, such as 2^10. This range of values, together with a time unit of 15/30 seconds are more than enough to describe the job behaviors also in very large environments. Note that a time unit of 60 second ease the job predictability from the administrators’ point of view.

The new scheduler gives administrators the possibility to “force” some system behaviors.

Really low starting priority values correspond to very important jobs and will ensure a quick
access to the highest priority queue.

High aging values define jobs that administrators do not want to be run too far from the planned start time.

Local highest priority values can specify background jobs which are run when system resource are under-utilized. The use of this directive is discouraged when associated to important jobs, since it can introduce job starvation.

The early start feature is effective when set for jobs of short duration, because they can efficiently use the system idle time without significantly delay high priority jobs.

Periodic jobs are easy to manage and improve the utilization of resources under situations of heavy system workload.

Blocked time windows allows administrators to organize the system workload and to manage specific client conditions. Time intervals can also be defined to inhibit the scheduler from running jobs, so to keep the system resources idle for administrative purposes.

Penalty time windows can be used to balance the system workload in periods where high usage of system resources are expected.

If history is available, the knowledge related to the duration of past jobs can be used to improve the backup plan.

Note that by setting the starting priority and the local highest priority directives to the same value an FCFS algorithm based on static priorities can be enabled.
7.5 Preliminary analysis in a real environment

The new scheduler has been deployed in one of the LNGS production chains since July 2012 and has been up and running for two months.

This system does not usually run under a heavy workload: about 50 jobs run on a daily basis on a storage infrastructure based on a tape library and a disk-based storage system. Job concurrency is possible up to two jobs per time.

These two months have not been enough to gather enough statistically relevant data for a detailed analysis of the benefits brought by the new scheduler utilization, but several improvements were seen.

These are:

- *Early start jobs* have been promoted for execution ahead of their planned start time in situations of idle system resources, helping to avoid workload peaks.

- *Periodic jobs* have regained regular execution in the system whenever long backup jobs introduced delays.

- *Job starvation* never arose.

- Due to system maintenance activities and software/hardware failures, the *recovery feature* has been used a few times and has always worked correctly.

- *Repeated failures* of the same jobs were very limited: in the few cases a job failed, it was rescheduled after a preset time interval and subsequent failures only happened in a few cases.

- A better *distribution of jobs* was possible by using time windows.

The system has shown few resource utilization peaks and the data collected about job waiting times is not yet enough for a measurement of the improvements achieved. Furthermore, a completely different and more suitable backup plan has been defined and it is difficult to compare the efficiency of the two schedulers two on different backup plans.

7. Analysis of the MLFQ scheduler
8. Conclusions

The new scheduling algorithm designed for the Bacula backup software is an MLFQ algorithm based on a dynamic priority strategy. The set of new directives introduced has made it possible to create a new scheduler capable of differentiating jobs into categories, and at the same time allow for an efficient queue management policy.

The benefits deriving from the application of the new scheduler are considerable:

- The new scheduler is able to effectively respond to the convoy effect situations.
- The aging mechanism makes sure that no job is indefinitely postponed, and prevents possible situation of job starvation.
- The dynamic priority strategy is more efficient than the original static priority system especially in environments where Bacula is capable of running concurrent jobs.
- The scheduler manages jobs so that the more important ones are rewarded more than low important ones. The dynamic priority and the aging concepts are the key to describe the importance of each job at runtime.
- The average waiting time of high priority jobs is reduced when compared to FCFS or static-priority algorithms. On the contrary, low priority jobs have associated higher average waiting times.
- The new directives defined for the job configuration give to administrators the possibility to define new job categories, and permits to the scheduler to perform better runtime decisions. Powerful job behaviors can be obtained by mixing together these directives.
- The schedule recovery feature is applicable after any short system interruption and guarantees continuity in the scheduling of jobs, even in the case of unexpected
The disadvantages of the new scheduler must also be mentioned:

- The main difficulty is related to the setup of the scheduler and the job configurations. The time interval for aging to apply and the dynamic priority range must be accurately defined, since they directly affect the definition of the starting priority and the aging directives for jobs in the backup plan.

- The scheduling algorithm complexity depends on the number of jobs in the active queues of the scheduler. The job screening made at the beginning of the day considerably reduces the number of jobs to monitor and therefore tries to maintain the system scalable.

In conclusion, the new scheduler solves most of the limits of the original scheduler, improves the job definition and modifies the original static priority strategy into a dynamic one. At the same time, the scheduler provides administrators with more control over the planning activity. More control has, as direct consequence, an higher possibility to achieve better utilization of the system resources.
8.1 Future works

There is no optimal scheduling algorithm applicable in every field. The requirements vary from domain to domain, and in the backup software domain some peculiar characteristics limit the efficiency of the scheduling algorithms. The unpredictability of the duration of jobs and the impossibility to preempt running jobs are only two of the problems related to the scheduling of jobs in this field. Domain-specific information can be used to improve the scheduling strategies.

In the proposed scheduling algorithm jobs are differentiated by categories. A further step would be to consider the job history and compute an average time required by each backup so that the duration of a job instance is predicted with a certain accuracy. Exponential average [19] can be used for this estimation, and this measure reused for filling idle resource times with jobs whose duration is expected to be within an idle time interval. This feature might significantly increase resource utilization in systems with low-medium loads. A similar approach is used in the Maui batch scheduler [7], [9] and referred to as backfilling.

Another improvement can be done considering the storage resources used by the backup system. If a storage daemon is based on tape libraries, the time required for the context switch between jobs (loading a different tape, wind the tape to the proper position, etc.) can be minimized by running one after the other the jobs which use the same tape, overriding the priority system.

A limit to the number of times a job is overcome by higher priority jobs can be also defined. This can be useful for avoiding the postponement of a job too many times.

In environments where multiple jobs can run concurrently, the performance of the scheduling algorithm must be studied so to evaluate alternatives to the priority-based scheduling system [7], [8], [9], [11].

A measurement of the job waiting time can be used to estimate the benefits that the new scheduler can achieve. This evaluation should be performed on similar backup plans and in a long interval of time (at least 6 months), preferably using two parallel backup systems.
8.2 Code evolution

The new scheduler is in use in a production environment at LNGS since July 2012. Its behavior is satisfactory and there is a general feeling of better usability and improved effectiveness.

Tests are still being run to find limitations and bugs.

The *Bacula team* has been informed about the new scheduler, and we are working to upstream the changes into one of the upcoming Bacula releases.
References


17. U. Schmid, J. Blieberger, (1992), Some investigations on FCFS scheduling in hard real

http://www.bacula.org/5.2.x-manuals/en/main/main.pdf [Last Access, September 28,
2012].


Acronyms

The acronyms used in this thesis are reported in the table 4.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNGS</td>
<td>Gran Sasso National Laboratory</td>
</tr>
<tr>
<td>INFN</td>
<td>Italian Institute for Nuclear Physics</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>GPL</td>
<td>General public license</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-in first-out</td>
</tr>
<tr>
<td>FCFS</td>
<td>First-come first-served</td>
</tr>
<tr>
<td>MLQ</td>
<td>Multilevel queue</td>
</tr>
<tr>
<td>MLFQ</td>
<td>Multilevel feedback queue</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
</tbody>
</table>

Table 4: Acronyms
Listing 1 reports the main procedure of the original Bacula scheduler.

The *find_runs* procedure is responsible for updating the *jobs_to_run* queue. Jobs that are expected to run in the current hour or the next are detected and inserted in the queue with a start time and priority sorting.

If the queue is still empty after an execution of the *find_runs* procedure, then the scheduler sleeps for an interval of time set to 60 seconds.

If the queue is not empty, the scheduler extracts the job in the first position and waits that the time for running the job arrives.

If the administrator updates the job configuration files and *reloads* the configurations [18] the scheduler is invalidated (*scheduled_invalidated*) and the schedule is recalculated.

```plaintext
main_procedure(){
    while (jobs_to_run → empty()){ 
        find_runs();
        if(jobs_to_run → empty())
            microsleep(scheduler_sleep_interval);
        else{
            next_job = jobs_to_run → first();
            while (not time to run next_job and not scheduled_invalidated){
                microsleep(scheduler_sleep_interval)
            }
            launch( next_job );
        }
    }
}
```

*Listing 1: Original scheduler pseudo-code: Main procedure*
APPENDIX B: New scheduler pseudo-code

This section provides the pseudo-codes of the new scheduler procedures.

1. MAIN PROCEDURE

Listing 2 reports the scheduling main procedure.

*Time_unit_interval* and *scheduler_sleep_interval* are two variables defined in the scheduler configuration, representing respectively the interval of time for the application of the aging procedure and the interval of time for the scheduler to wait before checking again for jobs to run.

The functions used are:

- *time()*: returns the current time of the system.
- *difftime()*: returns the difference between the input parameters of the function.
- *update_lists()*: updates queues by moving jobs, and updates the jobs dynamic priority values.
- *find_job()*: returns the best job to run, if any.
- *microsleep()*: pauses the scheduler main procedure.
- *launch()*: allocates resources to the job and runs it.
// variables

time_t now, last_time, time_unit_interval;

int time_units;

main_procedure()
{

    now = time ();
    time_units = difftime( now , last_time ) % time_unit_interval;

    if (time_units != 0) {
        update_lists(); // aging takes place inside this function
    }

    next_job = find_job();

    if (not next_job)
        microsleep(scheduler_sleep_interval);
    else
        launch(next_job);

    last_time = now;
}

Listing 2: New scheduler pseudo-code: Main procedure
2. UPDATE LISTS PROCEDURE

As time passes, the *update_lists* procedure is applied to maintain jobs in the most appropriate queue. When needed, jobs are moved to a different queue.

The aging procedure is invoked inside the *analyze_queue()* function. Dynamic priorities are updated depending on the time each job waits for the resource allocation and the jobs aging directives.

As time passes, three main situations can be detected by the scheduler.

CASE 1: DIFFERENT DAY

The day has changed.

The *inactive* queue must be analyzed and the job that are scheduled for running within the new day are extracted from this queue and moved to the most appropriate queue. The procedure continues with all the steps defined for case 2.

CASE 2: DIFFERENT HOUR

The hour has changed.

The *blocked* and the *penalized* queues are analyzed to check if time windows affecting jobs are over, in which case jobs are moved to the proper queue and, if necessary, the dynamic priorities are updated.

The *wake-up* procedure takes care of moving a job to the most appropriate queue. The *waiting*, the *early_start* and the *failed* queues are checked, and if a job status is changed, the procedure updates the location of the job and, if needed, its dynamic priority is updated.

The *update* procedure for the ready queue analyzes if time windows affect any of the jobs enqueued. If any, jobs are moved to the most appropriate queue. If a time window...
applies to a job, the job is moved to the *blocked* or the *penalized* queue, unless the job is in the *high priority* queue.

**CASE 3: SAME HOUR**

The day is not changed and the hour is the same as the last scheduler check.

The *blocked* queue does not need to be updated or analyzed since they are defined with time windows based on hour granularity.

The remaining queues are updated and aging is weighted and applied to each job according to the time unit elapsed since the last scheduler check.
update_lists() {

    switch(time_change) {

    case DIFFERENT_DAY:
        analyze_queue(inactive) → if (scheduled for this day) wake-up();

    case DIFFERENT_HOUR:
        analyze_queue(blocked) → if (not blocked) wake-up();
        analyze_queue(penalized) → if (not penalized) wake-up();
        analyze_queue(waiting) → if (time has arrived) wake-up();
        analyze_queue(early_start) → if (time has arrived) wake-up();
        analyze_queue(failed)
        analyze_queue(ready)
        break;

    case SAME_HOUR:
        analyze_queue(waiting) → if (time has arrived) wake-up();
        analyze_queue(early_start) → if (time has arrived) wake-up();
        analyze_queue(failed)
        analyze_queue(penalized)
        analyze_queue(ready)
        break;

    }
}

Listing 3: New scheduler pseudo-code: Update lists procedure
3. AGING PROCEDURE

The core of the aging procedure is in reported in listing 4.

For each job, the scheduler first computes the number of time_units to apply and then updates the dynamic priority of the job accordingly.

```plaintext
aging()
job → dynamicPriority = job → dynamicPriority – (job → aging * time_units);
return;
```

Listing 4: New scheduler pseudo-code: Aging procedure

4. FIND JOB PROCEDURE

At runtime, the scheduler identifies the most important job to run by calling the find_job procedure. This procedure is always executed after the job dynamic priorities have been updated.

The scheduler checks the high priority (hp) queue first. Although this queue is associated with a FCFS strategy, the scheduler still need to make sure that blocking time windows do not apply. The first non-blocked job is selected, if any.

If no jobs can be run from the hp queue, the local high priority (lhp), the ready and the penalized queues are analyzed and the best job among the three queue is extracted and returned to the main procedure.

If the scheduler still does not find a job, the early start queue is scanned and, among the set of
jobs that can be promoted for running, the job with the lowest starting priority value is returned, if any.

If no jobs are available for running, the procedure returns a null value.

Let $j_{lhp}^*$ be the job with the lowest dynamic priority among the job in the local high priority queue.

Let $j_{ready}^*$ be the job with the lowest dynamic priority among the job in the ready queue.

Let $j_{penal}^*$ be the job with the lowest dynamic priority among the job in the penalty queue.

Let $j_{es}^*$ be the job with the lowest dynamic priority among the jobs that can be promoted for an early start in the early_start queue.

The min() function selects the job with the lowest dynamic priority among the three, extracts it from the respective queue and returns it to the main scheduling procedure.
find_job() {
    while (not hp → empty()){
        if (not blocked)
            return job
    }
    if (not lhp → empty() ){
        analyze lhp queue
        set j_lhp*
    }
    if (not ready → empty() ){
        analyze ready queue
        set j_ready*
    }
    if (not penalized → empty() ){
        analyze penalized queue
        set j_penal*
    }
    if (not j_lhp* and not j_ready* and not j_penal*){
        if (not early_start → empty() ){
            analyze early_start queue
            set j_es*
            return j_es*
        }else { return null}
    }else{
        return min (j_lhp*, j_ready*, j_penal*)
    }
}

Listing 5: New scheduler pseudo-code: Find job procedure
5. SCHEDULE RECOVERY PROCEDURE

If the new scheduler is started with the schedule recovery procedure activated, the schedule is recovered accordingly to the pseudo-code reported in listing 6.

The recovery procedure distinguishes periodic jobs and fixed-time jobs.

If a job is periodic, the scheduler recovers the information related to its latest successful instance from the catalog database. This information is then reused to check if the job was supposed to run while the system was unavailable. If the job was supposed to run, the job is rescheduled according to the value defined in the reference directive and its dynamic priority updated.

Similar procedure applies for non periodic jobs. The most recent successful instance of a job is retrieved from the database and the successive planned start time analyzed. If the job was supposed to run, an instance is scheduled in the appropriate queue and its dynamic priority updated.

In both cases, if more than one job instance were supposed to run while the scheduler was down, the algorithm takes as a reference the oldest instance and applies aging to it, without scheduling duplicates of the same job.
schedule_recovery() {
    if (job_periodic) {
        switch (job_reference) {
            case planned_start:
                db_last_run = query_db_last_successful_instance()
                break;
            case actual_start:
                db_last_run = query_db_last_successful_instance()
                break;
            case last_completion:
                db_last_run = query_db_last_successful_instance()
                break;
        }
        if (!db_last_run) { reschedule() }
    } else {
        time_elapsed = difftime(now, db_last_run)
        if (time_elapsed > job_period) {
            realign_aging()
            reschedule_active()
        } else { reschedule_waiting() }
    }
}

Listing 6: New scheduler pseudo-code: Schedule recovery procedure
APPENDIX C: Backup software products web pages

The following tables report the official web pages of the backup software products cited in this thesis.

The last access to each web page dates back to September 28, 2012.

<table>
<thead>
<tr>
<th>Software</th>
<th>Official web page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bacula</td>
<td><a href="http://www.bacula.org/en/">http://www.bacula.org/en/</a></td>
</tr>
<tr>
<td>Amanda</td>
<td><a href="http://www.amanda.org/">http://www.amanda.org/</a></td>
</tr>
<tr>
<td>Rdiff-backup</td>
<td><a href="http://www.nongnu.org/rdiff-backup/">http://www.nongnu.org/rdiff-backup/</a></td>
</tr>
</tbody>
</table>

*Table 5: Free backup software products*

<table>
<thead>
<tr>
<th>Software</th>
<th>Enterprise</th>
<th>Official web page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup Exec</td>
<td>Symantec</td>
<td><a href="http://www.symantec.com/backup-exec">http://www.symantec.com/backup-exec</a></td>
</tr>
<tr>
<td>EMC NetWorker</td>
<td>EMC corporation</td>
<td><a href="http://www.emc.com/backup-and-recovery/networker/networker.htm">http://www.emc.com/backup-and-recovery/networker/networker.htm</a></td>
</tr>
<tr>
<td>IBM Tivoli Storage</td>
<td>IBM</td>
<td><a href="http://www.ibm.com/software/tivoli/">http://www.ibm.com/software/tivoli/</a></td>
</tr>
<tr>
<td>Manager</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NetBackup</td>
<td>Symantec</td>
<td><a href="http://www.symantec.com/netbackup">http://www.symantec.com/netbackup</a></td>
</tr>
<tr>
<td>Simpana</td>
<td>CommonVault</td>
<td><a href="http://www.commvault.com/simpana-software">http://www.commvault.com/simpana-software</a></td>
</tr>
</tbody>
</table>

*Table 6: Proprietary backup software products*