PROCEEDINGS OF SCIENCE

Scheduling and Load Balancing in the Parallel ROOT Facility (PROOF)

Gerardo Ganis
CERN
E-mail: Gerardo.Ganis@cern.ch

Jan Iwaszkiewicz*
CERN
Institute of Informatics, University of Warsaw
E-mail: Jan.Iwaszkiewicz@cern.ch

Fons Rademakers
CERN
E-mail: Fons.Rademakers@cern.ch

The Parallel ROOT Facility (PROOF) enables interactive analysis of distributed data sets in a transparent way. PROOF, which represents an alternative to batch-oriented distributed computing systems, is being successfully used by the PHOBOS experiment since a few years.
In view of the start-up of the LHC, PROOF underwent several developments and improvements. The main challenge was to adapt it to the large multi-user analysis environments. The ALICE collaboration has pioneered stress-testing of the system in the LHC environment, using a dedicated testbed, located in the CERN Computing Centre.
Among the developments for the LHC, load-balancing and resource scheduling play a central role. To optimize the work distribution, a new load-balancing strategy has been developed, with an adaptive algorithm to predict and circumvent potential data access bottlenecks.
Assignment of resources to users is addressed at two levels: centrally, by providing a scheduler with an abstract interface and locally, by implementing a mechanism to enforce resource quotas at worker level.

XI International Workshop on Advanced Computing and Analysis Techniques in Physics Research April 23-27 2007
Amsterdam, the Netherlands

*Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence.

PoS(ACAT)022

1. Introduction

The Parallel ROOT Facility (PROOF) [1] is an extension of the ROOT analysis framework [2] aimed at interactive ROOT-like analysis on a distributed system. PROOF represents an alternative to batch systems; by dynamically optimizing the use of resources it addresses in particular the case of short-and medium-duration jobs, for which the overhead from job preparation, submission and result merging may be substantial in batch systems. Although PROOF provides a framework to automatically parallelize any task consisting of a sequence of independent cycles (like Monte-Carlo simulations), it is being mostly used for the parallel analysis of large collection of ROOT files containing High-Energy Physics data stored in ROOT Tree format [2].

The system has already been described in detail [1]. For the purpose of this paper we just recall that PROOF realizes a multi-tier architecture, where *the client* is a ROOT session. It sends a job description to the (possibly composite) *master* which automatically parallelizes the job and distributes it among *the workers* (see fig. 1). The PROOF system works usually in connection with an XROOTD data serving system [3] running on the farm.

PoS(ACAT)022

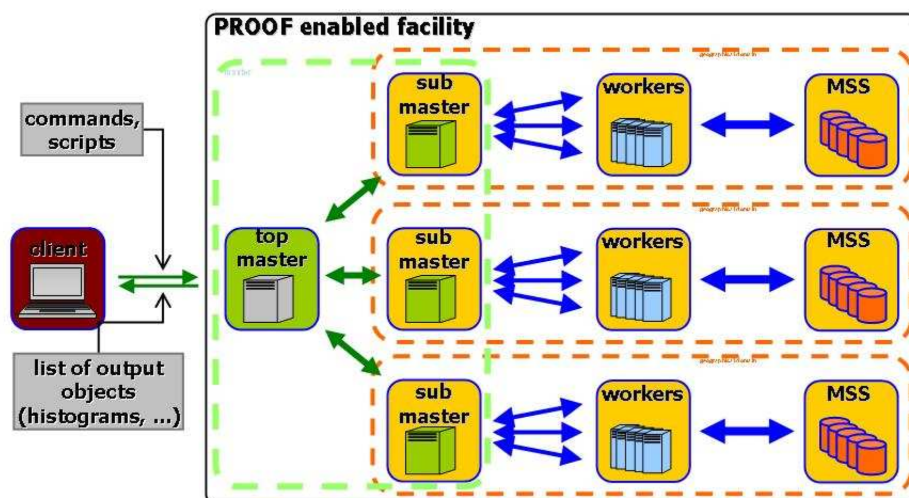

Figure 1: The PROOF multi-tier architecture. A ROOT session connects to the PROOF Master, which distributes the jobs to the workers. The workers may be connected to a Mass Storage System (MSS).

The PROOF system was originally developed as a joint effort between CERN and MIT and it is being used in production by the PHOBOS experiment [4]. This experience has shown that the system is suited for large clusters [5]. However the number of users in PHOBOS is small as compared to the groups of users in the LHC experiments [6]. One of the main challenges in preparing PROOF for the LHC analysis is to have the system correctly handling the available resources in

order to provide the best possible service to a large number of users performing different types of analysis.

The resource management in PROOF is split in two parts: *resource scheduling*, taking care of assigning the right share of resources to each job; *load-balancing*, optimizing the work distribution within jobs.

In the next section we review load-balancing and its engine (*the packetizer*), focusing on the latest improvements. In section 3 we present the main ideas and the current status of resource scheduling in the system. Finally, the paper is summarized in section 4.

2. Load-balancing using a Pull Architecture: *the packetizer*

The packetizer is responsible for load balancing a job between the workers assigned to it. It decides where each piece of work - called *packet* - should be processed.

An instance of the packetizer is created for each job separately on the master node. In case of a multi-master configuration, there is one packetizer created for each of the sub-masters. Therefore, while considering the packetizer, we can focus on the case of a single master without loosing generality.

The performance of the workers can vary significantly as well as the transfer rates to access different files. In order to dynamically balance the work distribution, the packetizer uses a *pull architecture (see fig. 2)*: when workers are ready for further processing they ask the packetizer for next packets.

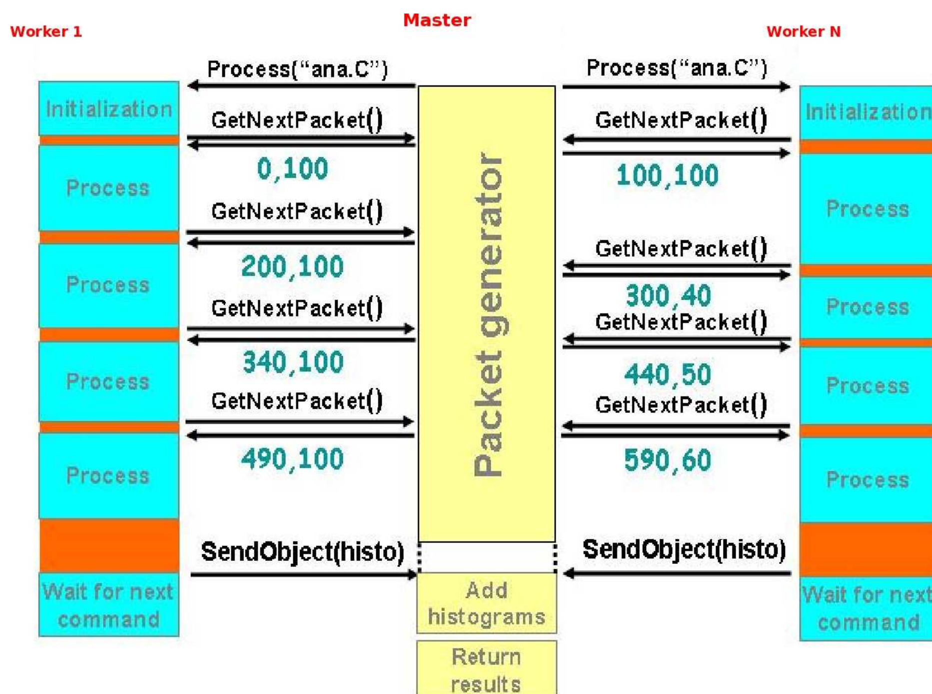

Figure 2: Pull architecture. The workers contact the packetizer, which is located on the master node. The packetizer distributes the work by assigning packets to be processed

PoS(ACAT)022

As PROOF is designed for interactive work, the job processing time is a critical parameter of the system. The packetizer has to distribute work in such a way that the job is finished as soon as possible, given the available resources. The optimal packetizer strategy is not unique but depends on the type of job so the system must be able to load different packetizers. In the rest of this section we focus on the available packetizers for data-driven analysis. Data sets for the data-driven jobs are sets of files with objects which are processed independently (e.g. ROOT Trees).

One of the main optimization criteria in this case is the data locality. In general, for a given job, a part of the data set to be processed is located on the worker nodes assigned to the job and the rest of data - on other non-worker nodes. In particular, either one of the parts can be empty.

2.1 The packetizer algorithms

The general algorithm of the data-driven packetizers consist of 3 parts. First the locations of all the files are checked. Then the files are validated. The validation is done in parallel by the workers and includes checking the number of entries in each file. During the processing phase the packetizer keeps creating packets and assigning them to the workers until the entire data set is processed. Below we describe strategies of assigning packets in two different packetizers.

Because having a large number of workers processing data from one physical hard drive may be counterproductive, there is an upper limit defined on the maximal number of workers reading from one server. It can be changed in order to adapt to the performance of the available hardware.

2.1.1 The basic packetizer

The strategy of the basic packetizer is determined by the fact that it is more efficient to analyze files at their location than to do it remotely. According to this strategy, packets are assigned in such a way that each worker first analyzes its local part of the data set (if it has one) and then processes remote files. The initialization includes calculating the base packet size, which is then equal for all the packets except for the last packets from each file. The strategy, detailed in appendix A.1, works very well if the file distribution is close to uniform.

2.1.2 The new adaptive packetizer

The main change with respect to the basic packetizer is the dynamic prediction of how much time each worker needs to process its local part of the data set. The predictions are used to balance the processing from the beginning of the job. Workers, which are expected to process their local data faster than the average, can be asked to process data from the other nodes also before their local processing is finished. The idea is based on the assumption that some of the nodes can become bottlenecks at the end of a job. The adaptive packetizer tries to identify the nodes and make sure that the data located there is processed at maximal possible speed in order to minimize a potential bottleneck effect. The packet size is becoming smaller towards the end of processing in order to avoid delay due to the long last packets. The algorithm is described in detail in appendix A.2.

2.2 Measuring the performance of the packetizers

To compare the performance of the two packetizer strategies outlined above, a cluster consisting of 34 dual processor machines (Intel Xeon 2.8 GHz) with 4 GB of RAM has been used. Each

PoS(ACAT)022

of the machines in the test has a single 250 GB hard drive connected by a SATA controller. [1] The tests were done using simulated Event Summary Data (ESD) of the ALICE experiment [7]. All the analyzed files were located on the available nodes.

The tool used to measure the performance of PROOF is an internal benchmark package [5]. The monitoring part of the package allows saving information about each packet and analyzing it after the job is processed.

The job processing time, which is minimized by the packetizer, depends on: *i)* the fraction of workers actively working throughout the job; *ii)* the average processing rate of the active workers, expressed in the number of events processed per second.

The benchmark package has been used to study the evolution in time of: *i)* the file access during the job, showing the number of workers having an opened file (*active* workers); *ii)* the processing rate per packet, classified according to one of three categories: data on a local file (*local packets*); data in a file on different active worker (*other-worker packets*); data in a file on a non-worker file node or on another remote server (*non-worker packets*).

2.3 Results

Performance measurement results for the basic packetizer are good in terms of efficiency and scalability for uniformly distributed data sets [5]. However, randomly distributed data sets were not processed so efficiently: when the data set distribution on the cluster was non-uniform, long tails appeared in the plots representing the instantaneous processing rate as well as the activity of the workers.

To see this in more detail we show an example of such a job processed with 16 workers (almost half of the cluster). When load-balanced by the basic packetizer, this job had the last part of processing (after about 900 s) done by only four workers (see Fig. 3a). This was due to the limitation on the number of concurrent workers described in section 2.1: after about 900 s all the remaining data was located on only one file server.

In Fig. 3b, we show the processing rates of all the packets of the job. One can notice that in the tail there are mostly remote packets. Also the processing rate is lower, which is due to the fact that four workers access files from the same physical hard drive.

In Fig. 4 we show the results obtained by processing the same job with the adaptive packetizer. In this case the processing of remote data starts earlier and it is done throughout the query; as a consequence, the long tails have disappeared. In Fig. 4a, one can notice that all the workers are active until the end of the job which means that the resources are not idle. For this job the total processing time has been reduced by about 30% when moving from the basic to the adaptive packetizer.

The prediction made by the adaptive packetizer allows very effective adjustment to the conditions in which the job is processed. Minimizing the long tail effects results in shorter processing time and better resource utilization.

[1] This cluster is intended to gradually become the planned ALICE Central Analysis Facility [7], consisting of about 500 CPUs and 200 TB of disk space.

PoS(ACAT)022

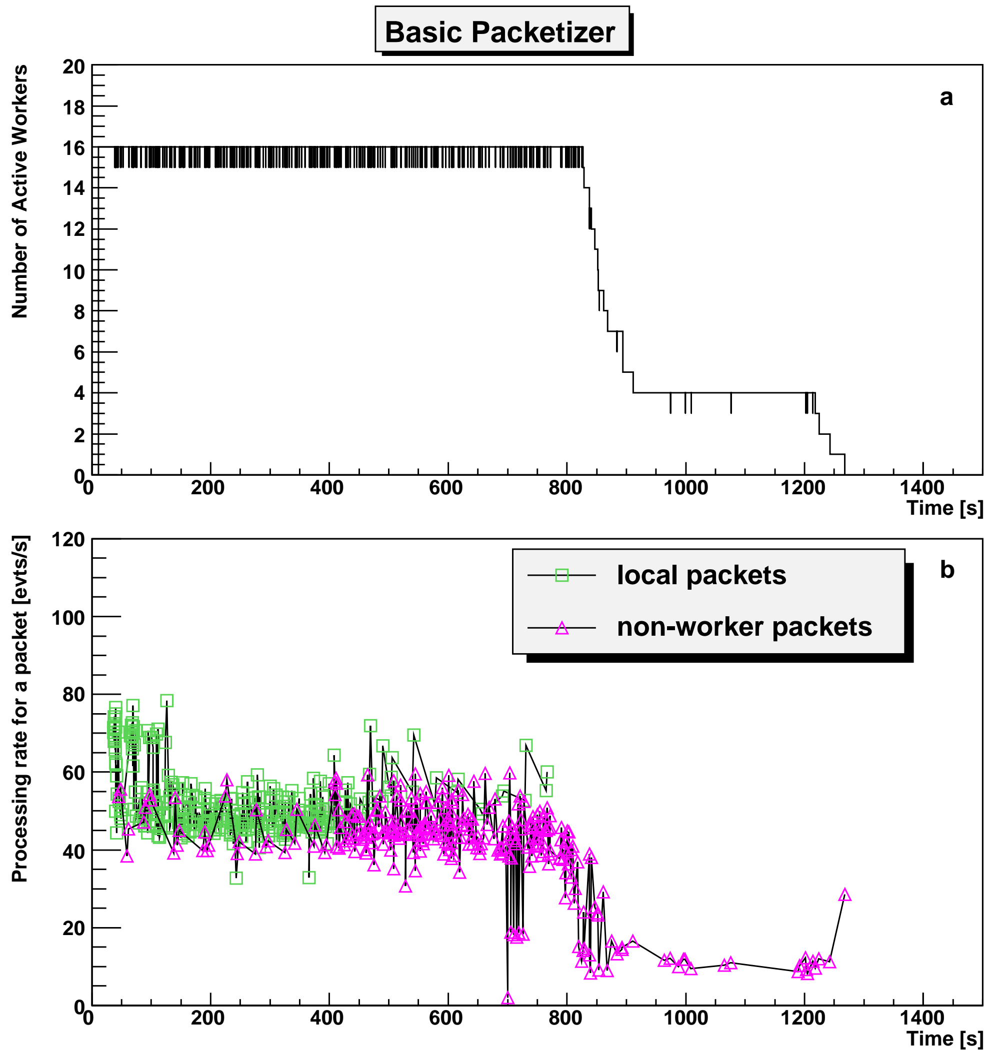

Figure 3: Time evolution analysis of an I/O intensive job processing an unevenly distributed data set using the basic packetizer; a) number of active workers; b) per-packet processing rates for data read from the local file system (*local packets*) and for data read from a non-worker node (*non-worker packets*).

3. Resource Scheduling

Resource scheduling is the task of assigning resources to the jobs in order to optimize the overall response of the cluster. The best performance is achieved by assigning the resources with the right balance between the two major factors: location of the data sets and the load of the machines. Resource scheduling allows also to enforce different *usage policies* , defining priorities or quotas of different users and groups.

PROOF implements resource scheduling at two levels. The first mechanism acts on every worker node. It controls the fraction of resources, used by each job, according to the user priority. The second level is a central scheduler which determines the user priorities and assigns workers to jobs.

PoS(ACAT)022

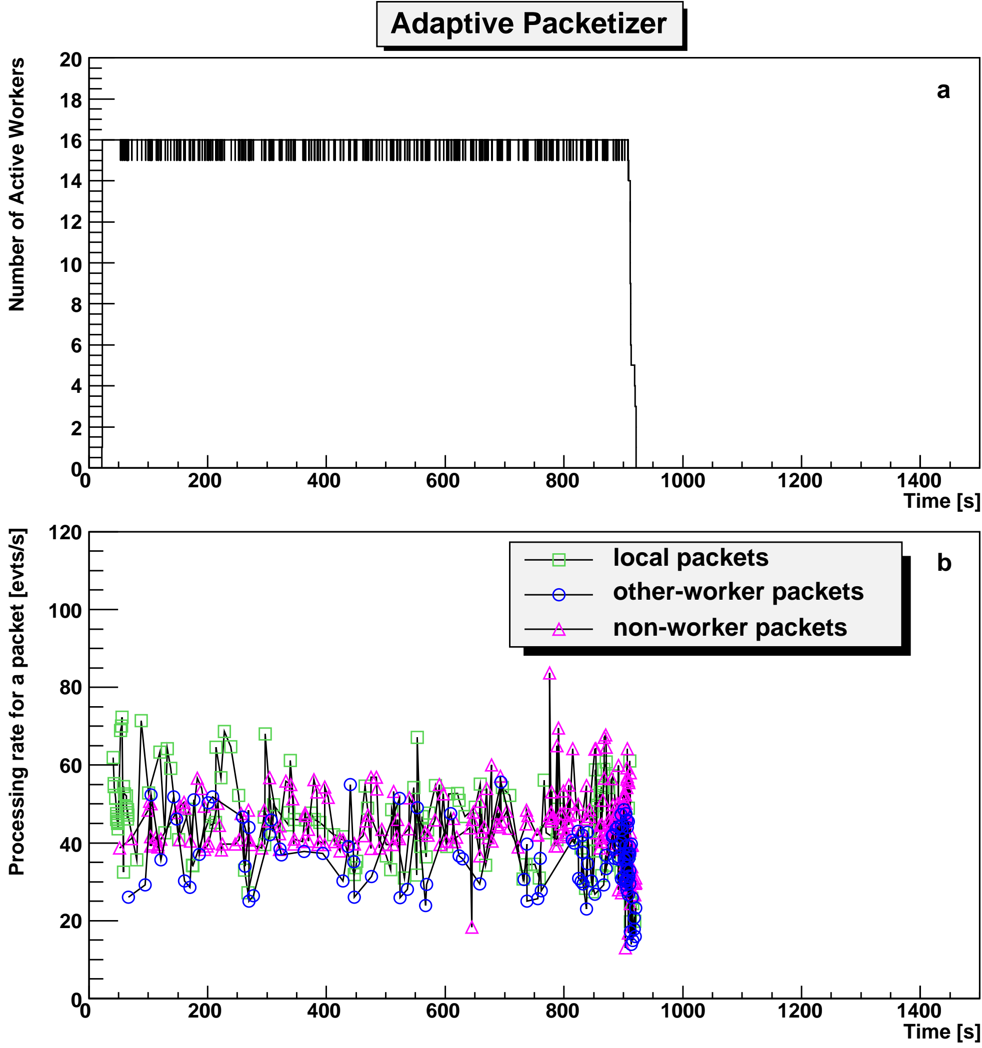

Figure 4: Time evolution analysis of an I/O intensive job processing an unevenly distributed data set using the basic packetizer; a) number of active workers; b) per-packet processing rates for data read from the local file system (*local packets*), for data read from another worker assigned to the job (*other-worker packets*), and for data read from a non-worker node (*non-worker packets*).

The mechanisms described in this section are still under development and are being tested by the ALICE experiment using the experimental testbed described in Sec. 2.2 .

3.1 Worker level resource control

Each worker node gets the group priorities from a file or from the master node. The active sessions are ranged according to their normalized priority $\bar{P}_j$ defined as

$$\bar{P}_j = \frac{P_j}{\frac{1}{N_{group}^{active}} \cdot \sum_i^{active} P_i} \cdot \frac{1}{N_{user;j}^{active}}$$

PoS(ACAT)022

with j the index of the group to which the session belongs, P_j the group priority, N^{active}_{group} the number of groups with active - i.e. processing - sessions, and $N^{active}_{user;j}$ the number of active users in j-th group [2]. The list of $\bar{P}_j$ values is further normalized in the range [1,40], in agreement with the values available in UNIX systems; each session is then *re-niced* using a *nice* value defined as

$$20 - \bar{P}_j^{[1,40]}$$

First tests done by ALICE have shown that this method works well (see Sec. 3.3); however, if the number of required priority levels is larger than 40, which may happen in the case of too many users in a low priority group, the groups with lower priority tend to get more resources than those targeted. To solve this problem, a mechanism of local round-robin job processing for users in the low priority groups is being prepared.

3.2 The central scheduler

The role of the central scheduler at job start-up is sketched in Fig. 5.

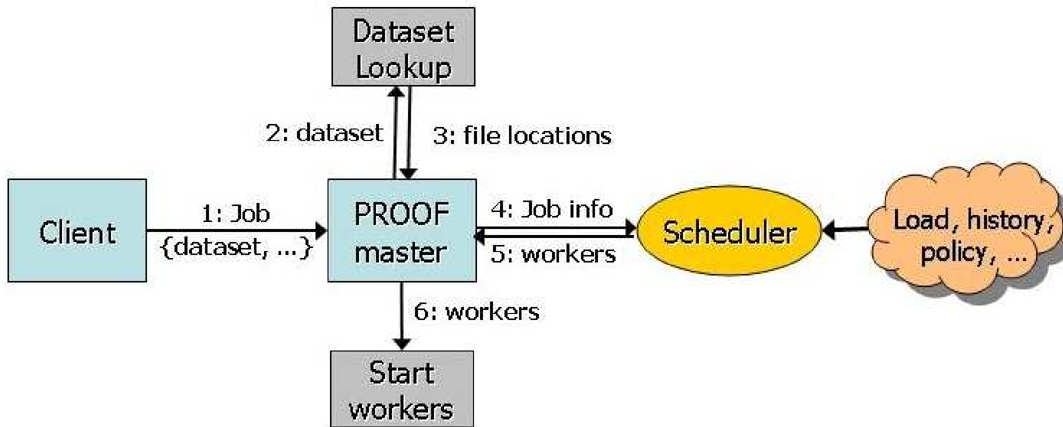

Figure 5: Starting a job with the central scheduler

The scheduler takes decisions according to the current state of the cluster and the usage history as well as the user and group priorities defined by the scheduling policy. Depending on the amount of active users, the scheduler assigns an appropriate number of worker nodes to each job in order to maintain the efficient processing and avoid overloading or congestion of any of the nodes.

For the case of fair-share or quota-based policies, the central scheduler may the place where the effective group and user priorities based on the cluster usage history are calculated.

The PROOF system interacts with the scheduler through an abstract interface; this allows different kinds of schedulers to be implemented, providing the required flexibility in the choice of the scheduler.

To allow fine-grained access to the information on usage of users and groups, PROOF can be configured to post the exact information about the resource usage of a given job through an

[2] An option to take into account the number of sessions of a given user is foreseen for the near future

PoS(ACAT)022

abstract monitoring interface; concrete implementations for MySQL [8] and MonALISA [9] are already available.

The PROOF basic scheduler implementation provides the possibility to assign the worker nodes based on the load and the group priority according to the simple formula:

$$N_{workers} = N_{CPU}^{free} \cdot f \cdot \bar{P}_j + N_{min}$$

where N_{CPU}^{free} is the number of free CPUs on the cluster nodes, f and N_{min} are, respectively, the fraction of free units and the minimal number of units to assign to a job, and $\bar{P}_j$ is the normalized priority previously defined.

3.3 Scheduling tests in ALICE: first results

The mechanism for worker level resource control described above has been tested in the ALICE setup described in Sec. 2.2. For this purpose the job resource information was posted to the ALICE MonALISA repository and a dedicated daemon, provided by ALICE, was used to calculate the effective group priorities based on the required policy. The resulting priorities were then feed in the basic central scheduler and communicated to the active sessions.

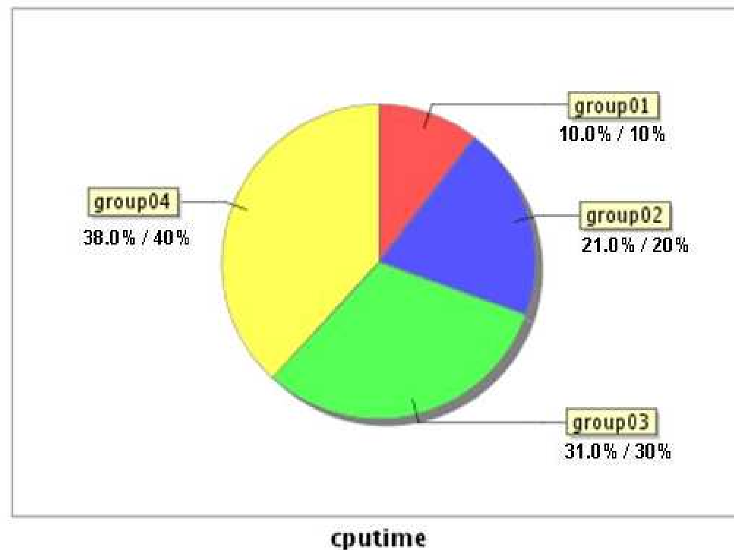

Figure 6: Results of CPU quota control test run by the ALICE experiment over one day. For each of the four groups the target (right) and achieved (left) CPU quotas are shown.

The results of a CPU quota control test run by the ALICE experiment over one day is shown in fig. 6. For this test, four groups were defined and assigned quotas ranging from 10 to 40%. Users in the groups were continously running typical ALICE analysis jobs. As it can be seen in the plot, the target quotas are reached within few percentage units.

PoS(ACAT)022

4. Summary

In this paper we discussed the techniques to manage resources which are being made available in PROOF in view of the LHC analysis.

Two different strategies to load-balance the work within a job have been described, including the recent improvements addressing the case of non-uniformly distributed data sets.

Resource assignment is controlled by a central scheduler, which is also determining the priorities of a given group or user, based on the policy defined by the experiment and the usage history; the priorities are enforced by a dedicated mechanism directly on the worker nodes.

These techniques for resource management are being tested by the ALICE collaboration using a dedicated cluster located at CERN. The first results are encouraging.

References

[1] M. Ballintijn *et al.*, *Parallel Interactive Data Analysis with PROOF*, in proceedings of ACAT 05, DESY, Zeuthen, Germany, Nucl.Instrum.Meth. A559 13-16 (2006).

[2] R. Brun and F. Rademakers, *ROOT - An Object Oriented Data Analysis Framework*, in proceedings of AIHENP'96 Workshop, Lausanne, Nucl.Instrum.Meth. A389 81-86 (1997).

[3] A. Dorigo, P. Elmer, F. Furano, A. Hanushevsky *XROOTD-A Highly scalable architecture for data access* WSEAS Transactions on Computers, 2005.

[4] M. Ballintijn, presentation at the CERN Application Area meeting, May 24, 2006; slides from http://indico.cern.ch/conferenceDisplay.py?confId=a062074 .

[5] M. Ballintijn, *et al.*, *Super scaling PROOF to very large clusters*, in proceedings of CHEP04, Interlaken, Switzerland.

[6] C. Eck, *et al.*, *LHC computing Grid : Technical Design Report*, CERN. Geneva. LHC Experiments Committee.

[7] F. Carminati, C.W. Fabjan, L. Riccati, H. de Groot, *ALICE computing : Technical Design Report, ALICE-TDR-12*, CERN-LHCC-2005-018.

[8] http://www.mysql.com

[9] http://monalisa.cacr.caltech.edu/monalisa.htm

PoS(ACAT)022

Appendix

A. Details of the packetizer algorithms

In this appendix we give more details on the algorithms of data-driven packetizers. The packetizers for data analysis use the same framework. The main difference is in the strategy of the GetNextPacket function. This function is called in response to the request from a worker for further processing. It assigns a part of file to the worker.

The general algorithm for a data-driven packetizer consists of the following three steps:

1. Check the exact locations of all the files.

2. If the data set is not validated, do validate it by opening all the files and checking the number of entries to be processed in each file (this step is done by workers in parallel).

3. Keep creating packets and assigning them to the workers until the entire data set is processed. The GetNextPacket function is assigning the packets.

A.1 Basic packetizer

A flow diagram of the GetNextPacket function in the basic packetizer is presented in fig. 7.

A.2 Adaptive packetizer

Fig. 8 represents the flow diagram of the GetNextPacket function in the adaptive packetizer. The decision whether to assign a local or remote file is taken based on dynamic prediction of the time that each worker needs to process its local part of the data set.

PoS(ACAT)022

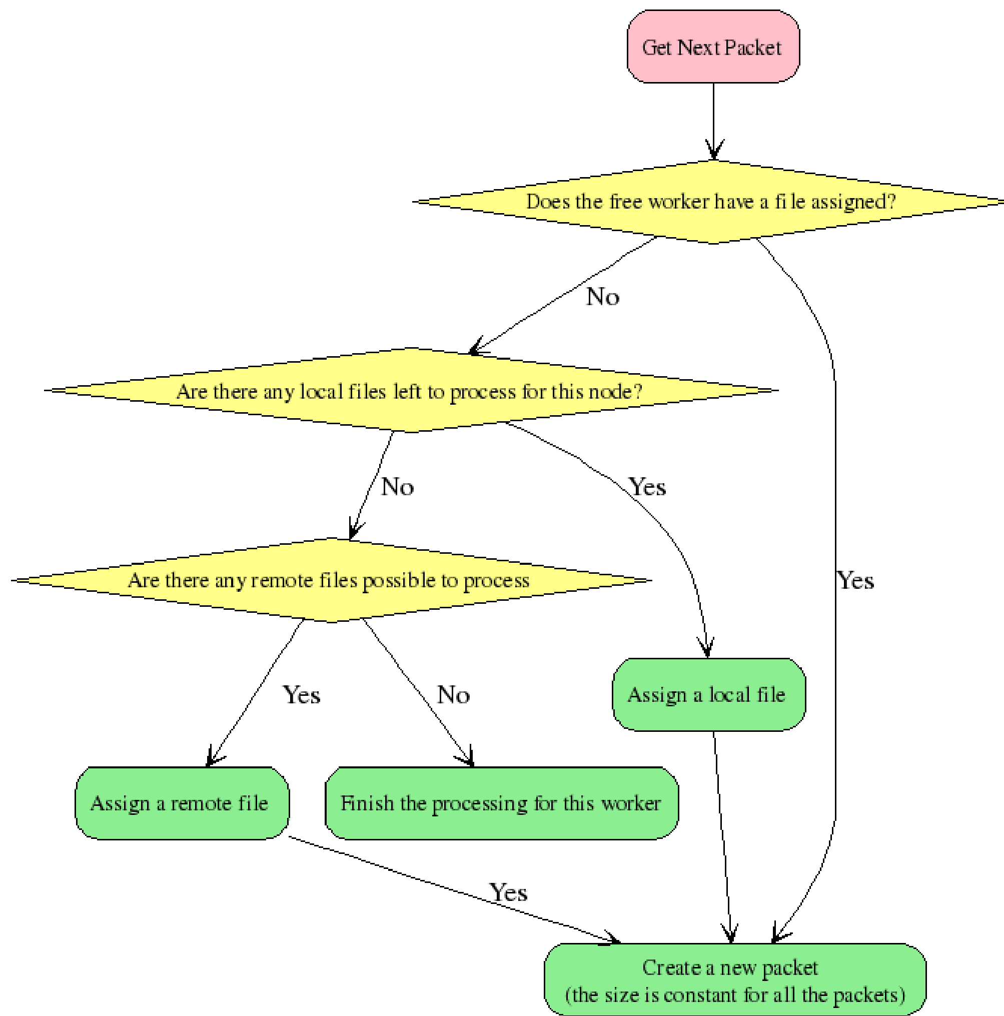

Figure 7: Flow diagram describing a simplified version of the GetNextPacket function algorithm in the basic packetizer.

PoS(ACAT)022

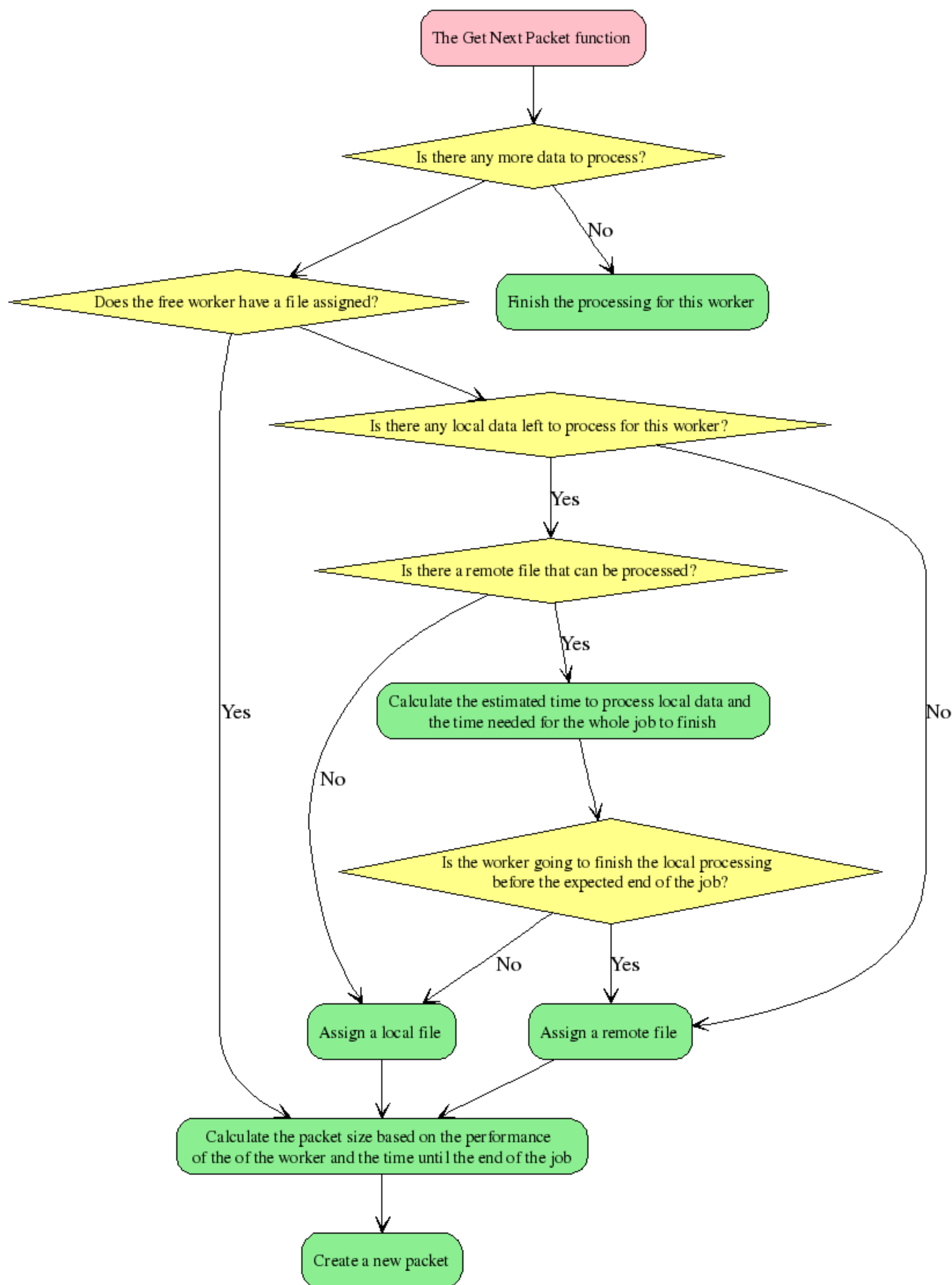

Figure 8: Flow diagram describing a simplified version of the GetNextPacket function algorithm in the adaptive packetizer.

PoS(ACAT)022